Université Alger 1

Rapport

Projet module "Bio-Inspired Computing" Implémentation d'un solveur SAT

Réalisé par :

BOUBEKEUR Fatma
Groupe : 02
Matricule : [redacted]

BOUDAH Maha
Groupe : 01
Matricule : [redacted]

2019-2020

Table des matières

Table des figures

Liste des tableaux

1. Présentation du problème SAT

SAT est une abréviation pour « boolean satisfiability problem », ou en français «problème de satisfaisabilité booléenne »[1]. C'est un problème de décision, qui consiste à vérifier si une formule booléenne est satisfiable. C'est-à-dire déterminer s'il existe une instanciation des variables propositionnelles qui rend la formule satisfiable.[2]

Exemple d'une formule SAT : $(x \vee y \vee z) \wedge (\neg x \vee \neg y) \wedge (\neg x \vee y \vee z)$

Une instance SAT est composé de :

Variables : Le problème SAT se compose d'un ensemble de variables, dans notre exemple on a trois variables : x, y et z qui peuvent prendre des valeurs 0 ou 1. Ces variables sont nommées «littéraux». Un littéral est une variable booléenne avec ou sans le connecteur de négation.

Clauses : Le problème SAT se compose d'un ensemble de clauses où chaque clause est une disjonction de littéraux. Dans notre exemple on 3 clauses : $(x \vee y \vee z)$ / $(\neg x \vee \neg y)$ / $(\neg x \vee y \vee z)$

L'opérateur « $\vee$ » exprime la disjonction booléenne tandis que l'opérateur « $\wedge$ » exprime la conjonction booléenne.

Dans le problème SAT, il est interdit d'avoir une variable et sa négation au sein d'une même clause.

Dans notre projet nous travaillerons avec une instance 3-SAT où toutes les clauses ont une longueur égale à 3.

La question posée par une instance SAT, est **« existe-t-il une instanciation des variables telle que la conjonction des clauses est vraie ? »**.

Selon la table de vérité présentée ci-dessous pour la formule propositionnelle de l'exemple précédent, les valeurs (par exemple, x = 1, y = 0, z = 1) sont des valeurs valides pour satisfaire la formule.

x	y	z	$x \vee y \vee z$	$\bar{x} \vee \bar{y}$	$\bar{x} \vee y \vee z$	F
0	0	0	0	1	1	0
0	0	1	1	1	1	1
0	1	0	1	1	1	1
0	1	1	1	1	1	1
1	0	0	1	1	0	0
1	0	1	1	1	1	1
1	1	0	1	0	1	0
1	1	1	1	0	1	0

Table 1.1 – Table de vérité

2. Objectif du projet

Le but de notre projet est de réaliser un solveur SAT qui permet de vérifier la satisfiabilité d'une formule propositionnelle en utilisant différentes techniques. Commençant par les méthodes aveugles : la recherche en largeur d'abord «BFS» et la recherche en profondeur d'abord «DFS». Passant ensuite à la recherche informée en utilisant l'algorithme A* avec le nombre de clauses satisfaites comme heuristique. On s'intéresse aussi à l'application des algorithmes évolutionnaires comme les algorithmes génétiques. Enfin, on teste l'algorithme de colonies de fourmis sur le problème SAT. Bien sûr on doit comparer les résultats de ces différents algorithmes.

3. Présentation du dataset « DIMACS CNF»

La représentation CNF est une représentation classique et c'est la première représentation d'un problèmes SAT. Il existe une base «DIMACS» où tous ces instances sont fournis au format CNF. Ce format est soutenu par la plupart des solveurs SAT dans la collection de solveurs de SATLIB.[3]

Les fichiers DIMACS sont composés par deux sections : les commentaires et les clauses.[3]

Un fichier DIMACS est un fichier texte (ASCII) qui commence par des lignes de commentaires marquées par la lettre minuscule «c», ils apparaissent généralement dans une section au début du fichier. Ces lignes de commentaires sont suivies de la ligne problème qui a le format suivant : «*p FORMAT NbrVariables NbrClauses*». p signifie ligne de problème, FORMAT contient le type de problème «cnf», NbrVariables contient un entier n spécifiant le nombre de variables dans le fichier, NbrClauses contient un entier m spécifiant le nombre des clauses dans le fichier. Les clauses figurent juste après la ligne problème et les variables sont supposées de 1 à n mais toutes les variables n'apparaissent pas nécessairement dans le fichier. Chaque clause est présentée par une séquence de nombres entiers séparé par des blancs, la négation d'une variable i est représenté par -i. Chaque clause se termine par 0.[3]

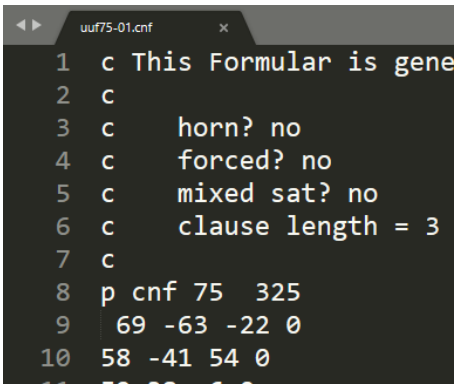

Figure 3.1 – Fichier DIMACS CNF

Notre dataset est composé de 100 fichiers. Chaque fichier contient 75 variables et 325 clauses où chaque clause a une longueur égale à 3.
Le fichier CNF de la figure correspond à la formule booléenne suivante :
$(x69 \vee \neg x63 \vee \neg x22) \wedge (x58 \vee \neg x41 \vee x54)$

4. Présentation des classes utilisées

La première étape à réaliser dans ce projet est de lire les fichiers qui se trouvent dans le répertoire du dataset,la classe *SATReader* du package *traitements.SAT* fait ce traitement. Ensuite, nous devons parser chaque fichier c'est à dire lire le fichier ligne par ligne pour obtenir les types de données significatifs. nous avons trois types de données essentiels dans notre projet : *Variable, Clause* et *NotVariable* qui s'étend de la classe *Variable*, la classe *SATParser* du package *traitements.SAT* est utilisée pour le parsing.

La vérification de la satisfiabilité des clauses de chaque fichier du dataset est effectuée dans la classe *SATChecker* du package *traitements.SAT*.

Deux structures de données essentielles sont utilisées dans ce projet :

- ***Variable[ ] variables*** : est un tableau de variables qui stocke toutes les variables du fichier.
- ***HashSet<Clause>*** : est une collection HashSet qui contient toutes les clauses du fichier.

La classe *SATChecker* contient plusieurs méthodes dans le but de résoudre le problème de satisfiabilité SAT. Parmi ces méthodes on cite :

- ***isSat()*** : pour vérifier si l'instance SAT est satisfiable, cela en calculant le nombre de clause satisfaites dans le fichier, si ce nombre est égale au nombre total des clauses dans le fichier donc TRUE l'instance est satisfiable sinon FALSE.

En ce qui concerne les parties dépendantes de notre problème nous avons :

1. **La fonction fitness :** ou la fonction d'évaluation, est une fonction qui nous permet d'évaluer la qualité d'une solution. Vu que notre problème est un problème de satisfiabilité, cette fonction sera une fonction de maximisation c'est-à-dire que la meilleur solution c'est celle qui aura satisfait le maximum des clauses (dans le cas idéal : 325 Clauses).
2. **Les opérateurs :** l'opérateur sera tout simplement une permutation de 0 à 1, c'est-à-dire changer la valeur du nœud courant, si ce dernier est à 0 il deviendra un 1 et vice versa
3. **La représentation de la solution :** la solution se présente sous la forme d'un vecteur qui contient un ensemble de variables qui satisfait l'ensemble des clauses.

Dans un premier temps, nous allons appliquer des méthodes exactes pour résoudre le problème SAT.

5. Partie 01 : Recherche aveugle et recherche informée (heuristique)

5.1 Algorithmes exactes (méthodes aveugles)

Les méthodes aveugles sont des méthodes qui réalisent une recherche exhaustive, ce sont des méthodes non informées qui parcourent toutes les solutions possibles sans utiliser des informations concernant la structure de l'espace d'états pour optimiser cette recherche.

5.1.1 La recherche en largeur d'abord

Cette méthode consiste à explorer les nœuds niveau par niveau, l'arborescence est construite par niveau et donc de manière horizontale. Le processus d'exploration consiste à explorer un nœud initial S0, puis ses successeurs, puis les successeurs non explorés des successeurs [4], la recherche est appliquée selon la stratégie FIFO (First In First Out).

Cette méthode garantit de trouver une solution si elle existe, mais elle est très gourmande en espace mémoire.

La complexité temporelle et spatiale pour cet algorithme est de O(S^a) où :

- **S** : est le nombre de sommets.
- **a** : est le nombre d'arcs.

5.1.2 La recherche en profondeur d'abord

Le parcours en profondeur (ou DFS, pour Depth First Search) est une méthode où l'expansion des nœuds les plus récemment engendrés s'effectue en premier. C'est la stratégie LIFO (Last In First Out), le nœud le plus profond est choisi pour l'expansion. Lorsqu'un certain seuil de profondeur est atteint, le processus considère le nœud du niveau précédent.[5]

La complexité temporelle et spatiale pour cet algorithme est de O(S^a) où :

- **S** : est le nombre de sommets.
- **a** : est le nombre d'arcs.

5.2 méthodes heuristiques

Une heuristique signifie 'aider à découvrir'. l'heuristique a pour rôle de diriger l'expansion des nœuds vers l'état cible en développant les nœuds les plus prometteurs. autrement dit, toute technique visant à accélérer la recherche est basée sur une information appelée heuristique. les méthodes heuristiques constituent par conséquent d'autres alternatives efficaces pour solutionner les problèmes complexes.[5]

Une heuristique dépend du problème à résoudre, donc elle nécessite une bonne maîtrise de ce dernier. Elle peut être pauvre, c'est à dire basée sur des informations relativement simples, dans ce cas les résultats seront moins efficaces ; comme elle peut être riche : basée sur des informations approfondis sur le problème, et sera dans ce cas beaucoup plus efficace mais difficile à implémenter.

L'utilisation d'une heuristique diminue l'espace des états, par conséquent accélérer la recherche et aboutir plus rapidement à une solution, mais cette dernière ne sera pas nécessairement la solution optimale.

5.2.1 L'algorithme A*

La recherche heuristique représente une technique alternative de résolution de problèmes en intelligence artificielle, largement utilisée pour les problèmes qui sont caractérisés par une explosion combinatoire d'états. Alors que les algorithmes non-informés réalisent une recherche systématique et aveugle, les techniques de recherche heuristiques utilisent un ensemble d'information qui permet d'évaluer la probabilité qu'un chemin allant du nœud courant au nœud cible soit meilleur que les autres.

Cette heuristique permet l'estimation des bénéfices des divers chemins avant de les parcourir et améliorer, dans la plupart des cas, le processus de recherche.

L'algorithme A* peut être considéré un cas particulier de l'algorithme de recherche ordonne. Il utilise la fonction heuristique f (aussi appelé fonction fitness).

Où n est un nœud représentant un état du problème, g(n) est le coût de la chaîne allant de l'état initial s à n et h(n) appelée heuristique est une estimation du coût de la chaîne reliant n à un état final. Cette heuristique estime le coût total de la chaîne entre l'état initial et un état cible qui passe par n.[5]

Dans notre cas la fonction fitness se définit comme suit :

$$f(n) = g(n) + h(n)$$

avec :

- n : un nœud représentant une valeur d'une variable.
- $g(n)$: est le nombre de fois que cette variable apparaît dans l'ensemble des clauses.
- $h(n)$: est le nombre de clauses satisfaites par cette valeur de variable.

5.3 Déroulement des algorithmes

La recherche en largeur d'abord

La figure ci-dessous présente le déroulement de l'algorithme de recherche en largeur (BFS) sur le problème SAT :

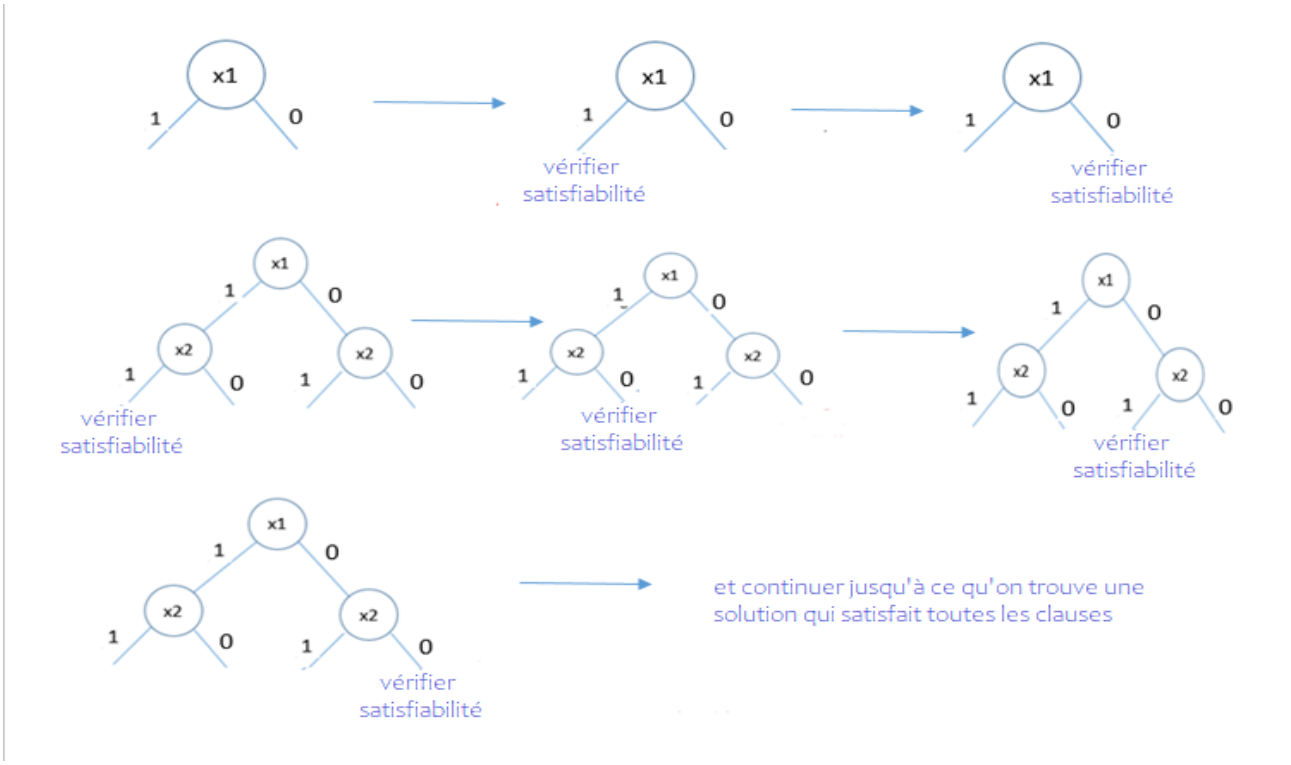

Figure 5.1 – Déroulement de l'algorithme BFS sur le problème SAT

La recherche en profondeur d'abord

La figure ci-dessous présente le déroulement de l'algorithme de recherche en profondeur (DFS) sur le problème SAT :

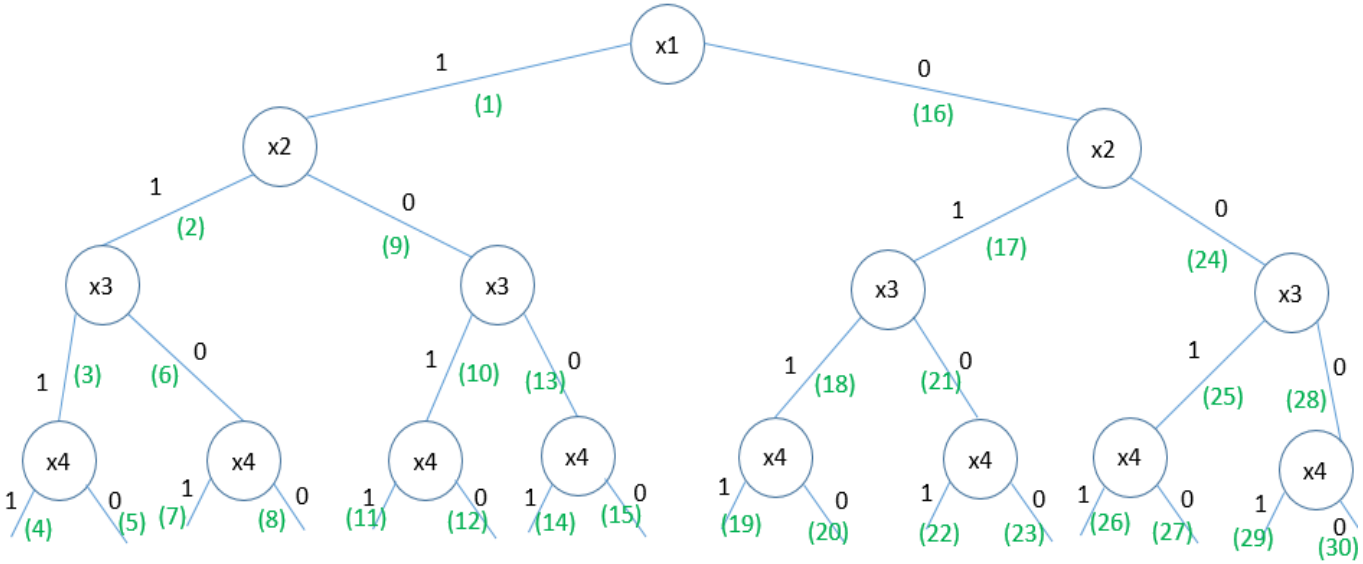

Figure 5.2 – Déroulement de l'algorithme DFS sur le problème SAT

L'algorithme A*

La figure ci-dessous présente le déroulement de l'algorithme A* sur le problème SAT :

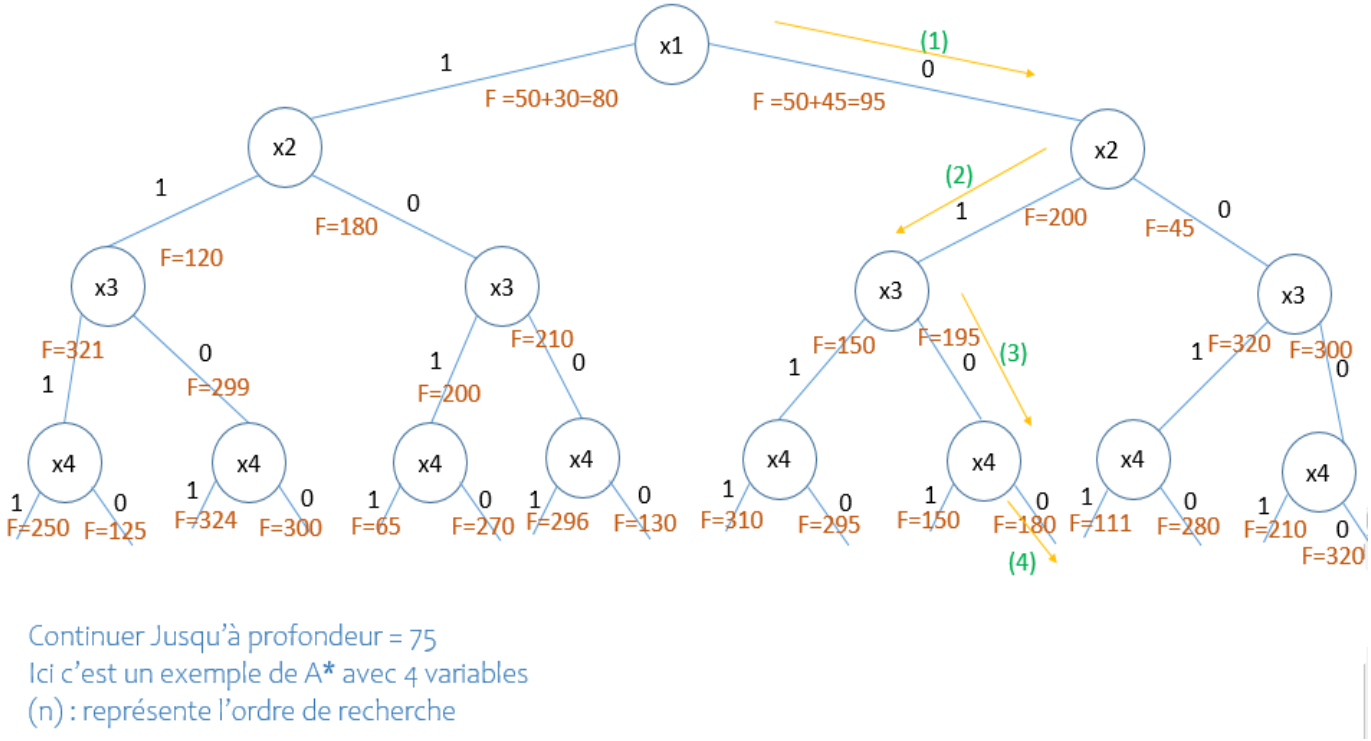

Figure 5.3 – Déroulement de l'algorithme A* sur le problème SAT

5.4 Les pseudo-codes

La recherche en largeur d'abord

La figure ci-dessous représente le pseudo-code de l'algorithme de recherche en largeur (BFS) :

Algorithme BFS

Entrée : X : ensemble de variables
C : ensemble de clauses
Sortie : Sol : ensemble de clauses

Début

Pour (i=1 à depthLimit) faire

Initialiser toutes les variables à 'true' ;

Tester si cette combinaison satisfait toutes les clauses

Si oui retourner cette combinaison sinon continuer ;

Générer les successeurs et tester les combinaisons de variables de gauche à droite ;

Si parmi ces successeurs il existe une combinaison qui satisfait toutes les clauses ;

Retourner cette combinaison de variables sinon continuer ;

Fait ;

Fin

Figure 5.4 – Pseudo-code de l'algorithme BFS

La recherche en profondeur d'abord

La figure ci-dessous représente le pseudo-code de l'algorithme de recherche en profondeur (DFS) :

```
Algorithme DFS

Entrée :  X : ensemble de variables
          C : ensemble de clauses
Sortie :  Sol : ensemble de clauses
Var :     depth : entier //le seuil de la profondeur
          variableStack : pile de variables

Début
    Empiler le nœud initial dans la pile variableStack ;
    Si la valeur de cette variable satisfait toutes les clauses alors retourner cette variable ;
    Sinon continuer ;
    Tant que (variableStack est non vide) faire
        Dépiler n le premier nœud de la pile ;
        Si (profondeur de l'arbre <> depth) alors
            Déterminer les successeurs de n et les empiler dans variableStack;
            Créer un chaînage de ces nœuds vers n;
            Si parmi ces successeurs il existe une combinaison qui satisfait toutes les clauses alors
            Succès : retourner la solution ; fsi ;
        Fsi ;
    Fait ;
    Retourner 'false' ;  //tout l'arbre est parcouru sans arriver à une solution;
Fin
```

Figure 5.5 – Pseudo-code de l'algorithme DFS

L'algorithme A*

La figure ci-dessous représente le pseudo-code de l'algorithme A* :

```
Algorithme A*
Entrée : X : ensemble de variables
         C : ensemble de clauses
         H : fonction heuristique
Sortie : Sol : ensemble de clauses
Var :    variableStack : file de variables
Début
    Empiler le nœud initial dans la pile variableStack qui a la plus grande valeur de f (la meilleur fitness) ;
    Si la valeur de cette variable satisfait toutes les clauses alors retourner cette variable ;
    Sinon continuer ;
    Tant que (variableStack est non vide) faire
        Retirer n le premier nœud de la file qui a la plus grande valeur de f (la meilleur fitness) ;
        Si n satisfait toutes les clauses alors retourner cette valeur ; la solution est un chainnage
        arrière  de ce nœud vers la racine ;
        sinon
            si n a des successeurs alors
                pour chaque ni dans n faire
                    f(ni)= g(ni) + h(ni) ;
                    enfiler ni avec f(ni) dans la file selon f ;
                    créer un chainage de ni vers n ;
                fait ;
            fsi ;
        fsi ;
    fait ;
Fin
```

Figure 5.6 – Pseudo-code de l'algorithme A*

5.5 Complexité des Algorithmes

1. **La recherche en largeur d'abord :**
   - Complexité temporelle : O (2^{75})
   - Complexité spatiale : O (2^{75})
2. **La recherche en profondeur d'abord :**
   - Complexité temporelle : O (2^{75})

— Complexité spatiale : O (2^{75})

3. **L'algorithme A* :**

— Complexité temporelle : O (2^{75})

— Complexité spatiale : O (2^{75})

2 représente le nombre d'arcs (nombre de successeurs) et 75 est le nombre de variables.

5.6 Analyse des résultats

Nous allons analyser les résultats obtenus lors de l'exécution des trois algorithmes vu précédemment.

Étant donné que le dataset contient 100 instances, nous avons sélectionnés aléatoirement 10 instances pour pouvoir effectuer une analyse des résultats.

5.6.1 Environnement expérimental

L'exécution de ces algorithmes s'est effectuée sur un PC de :

— Ram : 4 GO.

— Processeur : Intel(R) Core™ i5-3210M CPU @ 2.00 GHz 2.60 GHz.

— Système d'exploitation : Windows 10-64 bits.

— Langage de programmation : JAVA.

— Environnement de développement : Eclipse.

5.6.2 La recherche en largeur d'abord

Le tableau ci-dessous représente le taux de satisfiabilité des 10 fichiers sélectionnés aléatoirement des deux benchmarks UF-75 et UUF-75 avec un algorithme de parcours par largeur (BFS).

benchmark	Nom du fichier	Nombre de clauses satisfaites	pourcentage
UF-75	uf75-09.cnf	300	92.31%
	uf75-081.cnf	295	90.77%
	uf75-08.cnf	291	89.54%
	uf75-079.cnf	295	90.77%
	uf75-078.cnf	279	85.85%
	uf75-07.cnf	296	91.08%
	uf75-069.cnf	290	89.23%
	uf75-062.cnf	283	87.08%
	uf75-048.cnf	294	90.46%
	uf75-029.cnf	298	91.69%
	Total		89.88 %
UUF-75	uuf75-099.cnf	285	87.69%
	uuf75-089.cnf	293	90.15%
	uuf75-076.cnf	282	86.77%
	uuf75-070.cnf	292	89.85%
	uuf75-066.cnf	293	90.15%
	uuf75-059.cnf	297	91.38%
	uuf75-042.cnf	290	89.23%
	uuf75-039.cnf	299	92%
	uuf75-031.cnf	286	88%
	uuf75-027.cnf	292	89.85%
	Total		89.50 %

Table 5.1 – Résultats de la recherche BFS sur UF-75 et UUF-75

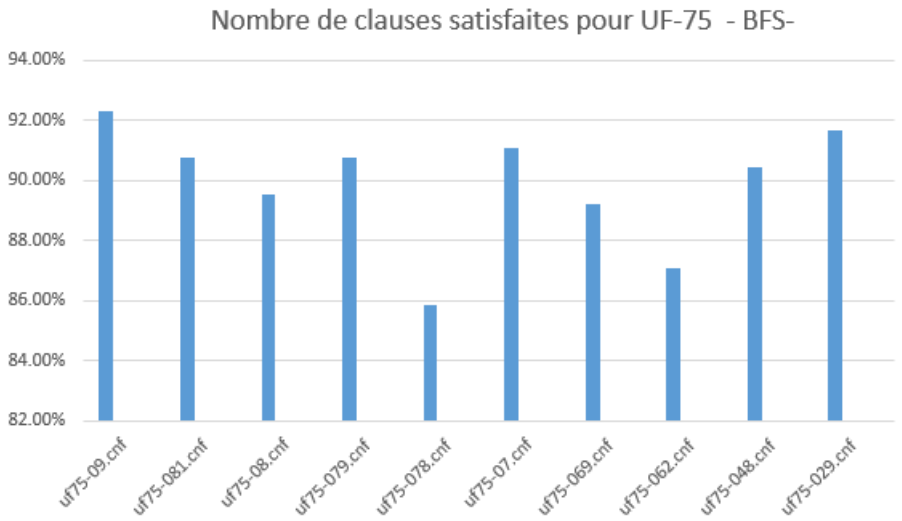

Figure 5.7 – Histogramme montrant les résultats de l'application de l'algorithme BFS pour le problème SAT sur les fichiers de Benchmarks UF-75

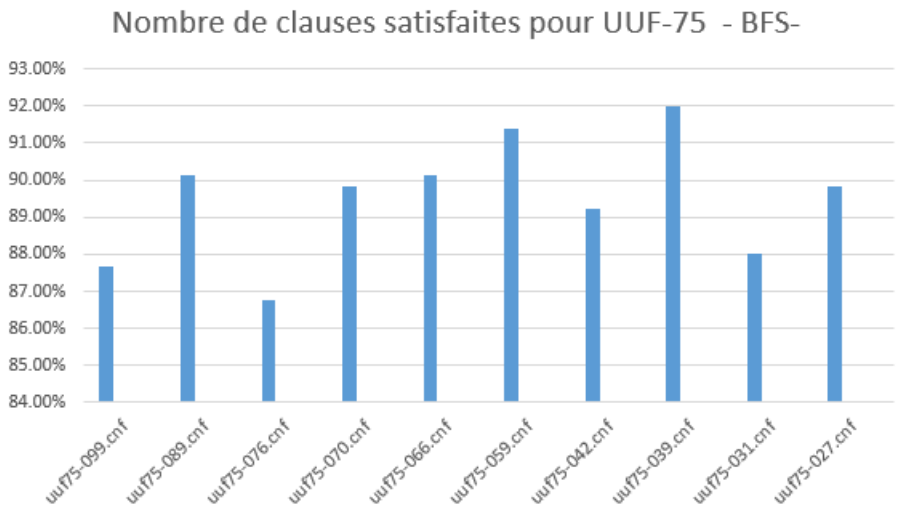

Figure 5.8 – Histogramme montrant les résultats de l'application de l'algorithme BFS pour le problème SAT sur les fichiers de Benchmarks UUF-75

5.6.3 La recherche en profondeur d'abord

Le tableau ci-dessous représente le taux de satisfiabilité des 10 fichiers sélectionnés aléatoirement des deux benchmarks UF-75 et UUF-75 avec un algorithme de parcours par profondeur (DFS).

benchmark	Nom du fichier	Nombre de clauses satisfaites	pourcentage
UF-75	uf75-09.cnf	296	91.08%
	uf75-081.cnf	293	90.15%
	uf75-08.cnf	277	85.23%
	uf75-079.cnf	292	89.85%
	uf75-078.cnf	275	84.61%
	uf75-07.cnf	290	89.23%
	uf75-069.cnf	288	88.62%
	uf75-062.cnf	277	85.23%
	uf75-048.cnf	283	87.08%
	uf75-029.cnf	290	89.23%
	Total		88.03 %
UUF-75	uuf75-099.cnf	273	84%
	uuf75-089.cnf	290	89.23%
	uuf75-076.cnf	281	86.46%
	uuf75-070.cnf	284	87.38%
	uuf75-066.cnf	290	89.23%
	uuf75-059.cnf	285	87.69%
	uuf75-042.cnf	283	87.08%
	uuf75-039.cnf	293	90.15%
	uuf75-031.cnf	282	86.76%
	uuf75-027.cnf	290	89.23%
	Total		87.42 %

Table 5.2 – Résultats de la recherche BFS sur UF-75 et UUF-75

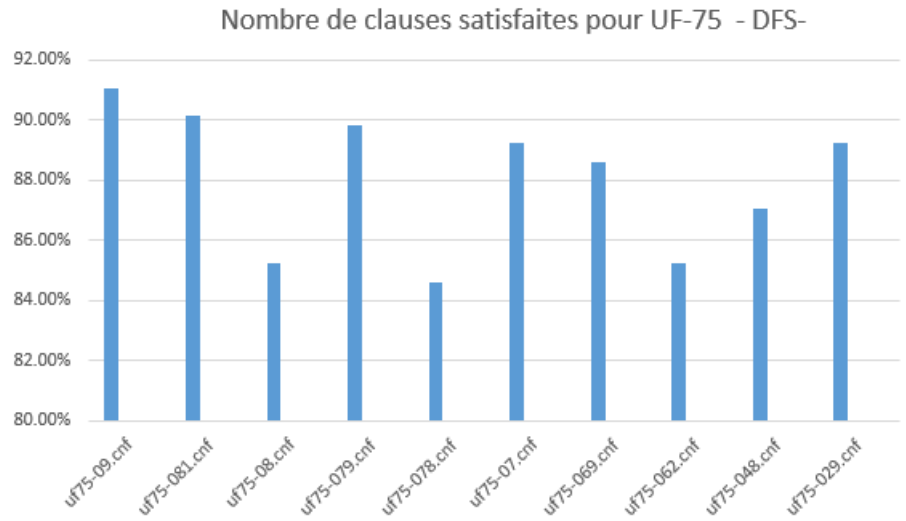

Figure 5.9 – Histogramme montrant les résultats de l'application de l'algorithme DFS pour le problème SAT sur les fichiers de Benchmarks UF-75

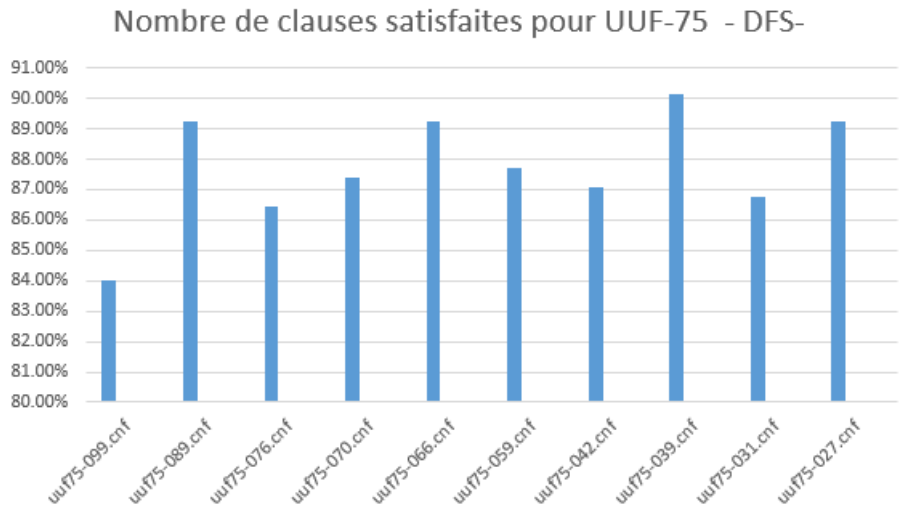

Figure 5.10 – Histogramme montrant les résultats de l'application de l'algorithme DFS pour le problème SAT sur les fichiers de Benchmarks UUF-75

5.6.4 L'algorithme A*

Le tableau ci-dessous représente le taux de satisfiabilité des 10 fichiers sélectionnés aléatoirement des deux benchmarks UF-75 et UUF-75 avec un algorithme A*.

benchmark	Nom du fichier	Nombre de clauses satisfaites	pourcentage
UF-75	uf75-09.cnf	320	98.46%
	uf75-081.cnf	318	97.85%
	uf75-08.cnf	317	97.54%
	uf75-079.cnf	313	96.31%
	uf75-078.cnf	309	95.08%
	uf75-07.cnf	317	97.54%
	uf75-069.cnf	311	95.69%
	uf75-062.cnf	312	96%
	uf75-048.cnf	315	96.92%
	uf75-029.cnf	318	97.85%
	Total		96.92 %
UUF-75	uuf75-099.cnf	310	95.38%
	uuf75-089.cnf	310	95.38%
	uuf75-076.cnf	311	95.69%
	uuf75-070.cnf	316	97.23%
	uuf75-066.cnf	312	96%
	uuf75-059.cnf	312	96%
	uuf75-042.cnf	315	96.92%
	uuf75-039.cnf	317	97.54%
	uuf75-031.cnf	307	94.46%
	uuf75-027.cnf	312	96%
	Total		96.06 %

Table 5.3 – Résultats de la recherche BFS sur UF-75 et UUF-75

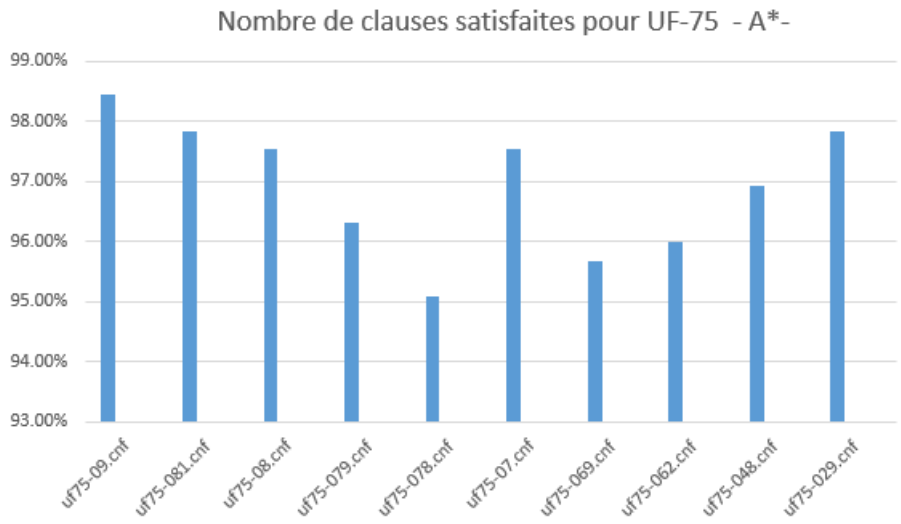

Figure 5.11 – Histogramme montrant les résultats de l'application de l'algorithme BFS pour le problème SAT sur les fichiers de Benchmarks UF-75

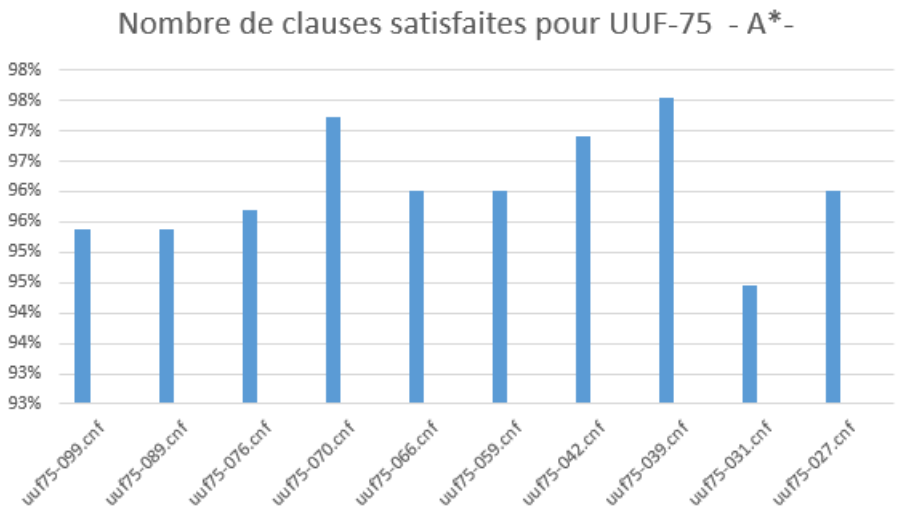

Figure 5.12 – Histogramme montrant les résultats de l'application de l'algorithme A* pour le problème SAT sur les fichiers de Benchmarks UUF-75

5.6.5 Discussion des résultats

Après l'analyse des résultats des tableaux et des graphes des trois algorithmes précédents, nous pouvons remarquer qu'aucun algorithme n'a atteint la solution la plus optimale c'est-à-dire celle qui satisfait toutes les clauses. néanmoins nous avons pu arriver a une solution très proche avec l'algorithme A*, en effet ce dernier a un pourcentage de satisfiabilité égal à 96.92 % pour le benchmark UF-75 et 96.06 % pour UUF-75. Tandis que les pourcentages de l'algorithme BFS sont de 89.88% et 89.50% pour UF-75 et UUF-75 respectivement. Quant à DFS les pourcentages sont de 88.03% et 87.42%.

On peut conclure que l'ordre d'efficacité de ces trois algorithmes du mieux efficace au mois efficace est comme suit :

1. A* (efficacité moyenne = 96.5%)
2. BFS (efficacité moyenne = 89.7%)
3. DFS (efficacité moyenne = 87.73%)

6. Partie 02 : les algorithmes génétiques

6.1 Historique

— Les algorithmes génétiques ont été inventés par Jonh Holland dans les années 60.[6]

— Repris notamment par Golberg dans les années 70, Le principe des algorithmes génétiques s'inspire directement des lois de la sélection naturelle, décrites par Darwin.[6]

Cette technique connait aujourd'hui un grand succès. On l'utilise dans la résolution de problèmes complexes, nécessitant des temps de calcul élevés.

6.2 Algorithmes génétiques

Les algorithmes génétiques peuvent être décrits comme une méthode heuristique basée sur la «Survie du plus apte» (Survival of the fittest). C'est un outil utile pour les problèmes de recherche et d'optimisation. L'algorithme génétique soulève quelques caractéristiques importantes : c'est d'abord un algorithme stochastique ; l'aspect aléatoire joue un rôle essentiel, la sélection et le croisement nécessitent des procédures aléatoires. Un deuxième point très important est que les algorithmes génétiques considèrent toujours une population de solutions : garder en mémoire plus d'une solution unique à chaque itération offre de nombreux avantages : l'algorithme peut combiner de différentes solutions pour en obtenir de meilleures et ainsi, il peut utiliser les avantages de l'assortiment.[7]

Les algorithmes génétiques appartiennent à la famille des algorithmes évolutionnistes. Leur but est d'obtenir une solution approchée à un problème d'optimisation, lorsqu'il n'existe pas de méthode exacte (ou que la solution est inconnue) pour le résoudre en un temps raisonnable.[8]

6.3 Principe des algorithmes génétiques

6.3.1 Terminologies génétiques :

Dans les algorithmes génétiques nous retrouvons les notions importantes suivantes : Codage d'une solution, fonction d'évaluation, population, individu et gène.

Codage d'une solution : nous représenterons une solution par un tableau de booléen (false $\longrightarrow$0 et true $\longrightarrow$1) de taille 75.

[false,false,false,true,true,false,false,true,false,true,true,false,true,false,true,false,true, ...]

Fonction d'évaluation : nous allons évaluer les solutions selon le nombre maximum de clauses satisfaites par chaque solution.

Population : est un ensemble de solutions donc c'est un ensemble de tableau de variables de taille M=100.

Individu ou chromosome : représente une solution au problème dans notre cas c'est un vecteur de variables.

Gène : est une caractéristique, une particularité. Dans notre cas un gène représente une variable dont la valeur peut être true ou false.

6.3.2 Les opérateurs des algorithmes génétiques :

Il y a trois opérateurs d'évolution dans les algorithmes génétiques :[9]

- ***La sélection :*** Choix des individus les mieux adaptés.[9]
- ***Le croisement :*** Mélange par la reproduction des particularités des individus choisis.[9]
- ***La mutation :*** Altération aléatoire des particularités d'un individu.[9]

La sélection

Consiste à choisir les individus les mieux adaptés afin d'avoir une population de solution la plus proche de converger vers l'optimum global.[9]

Il existe plusieurs techniques de sélection. Voici les principales utilisées :[9]

- ***Sélection par rang :*** cette technique de sélection choisit toujours les individus possédant les meilleurs scores d'adaptation.[9]
- ***Probabilité de sélection proportionnelle à l'adaptation :*** technique de la roulette ou roue de la fortune, pour chaque individu, la probabilité d'être sélectionné est proportionnelle à son adaptation au problème.[9]
- ***Sélection par tournoi :*** cette technique utilise la sélection proportionnelle sur des paires d'individus, puis choisit parmi ces paires l'individu qui a le meilleur score d'adaptation.[9]
- ***Sélection uniforme :*** la sélection se fait aléatoirement, uniformément et sans intervention de la valeur d'adaptation.[9]

Nous avons opté pour une sélection par rang (ranking) qui ne fait pas intervenir le hasard, contrairement à la roulette, car elle choisit les n meilleurs individus de la population. Chaque individu a ainsi une probabilité de sélection dépendant de son évaluation, avec une plus forte probabilité pour les meilleurs individus. Nous avons sélectionnés les 50 meilleurs individus de la population.

Voici un exemple avec des individus en représentation booléenne une fois la sélection effectuée :

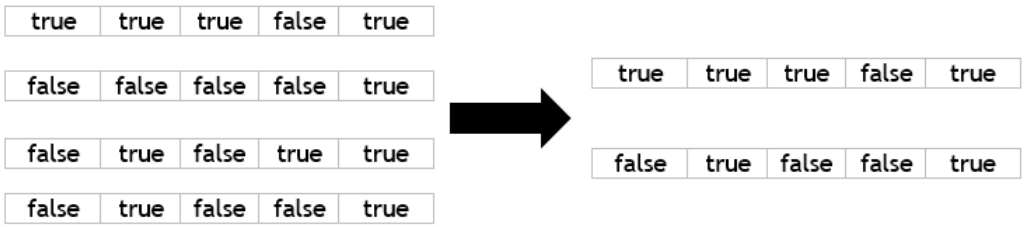

Figure 6.1 – Exemple de sélection

On réalise ensuite un croisement entre deux chromosomes parmi la population restante.

Croisement

Celui-ci permet le brassage génétique de la population et l'application du principe d'hérédité de la théorie de Darwin. Il existe deux méthodes de croisement : simple ou double enjambement.[9]

Le simple enjambement consiste à fusionner les particularités de deux individus à partir d'un pivot, afin d'obtenir un ou deux enfants :[9]

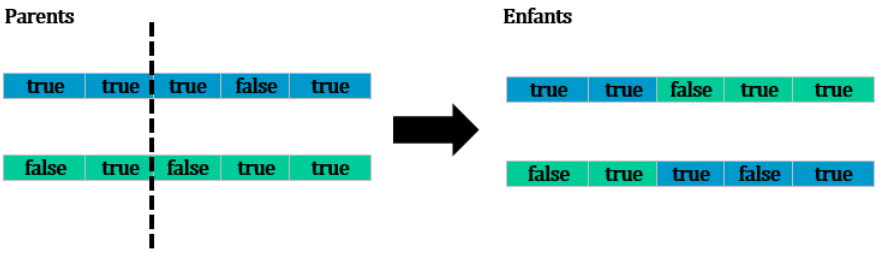

Figure 6.2 – Exemple de croisement en simple enjambement

Le double enjambement repose sur le même principe, sauf qu'il y a deux pivots :[9]

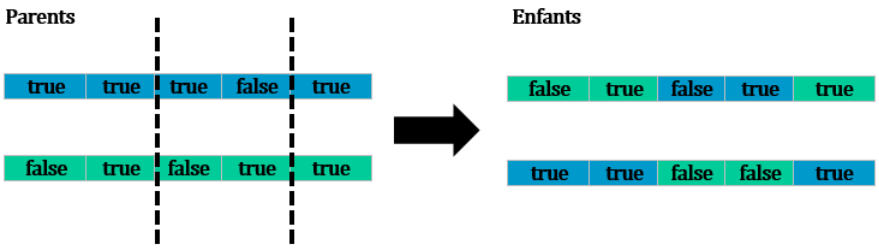

Figure 6.3 – Exemple de croisement en double enjambement

On réalise ensuite une mutation sur les enfants obtenues lors du croisement.

La mutation

La mutation consiste à altérer un gène dans un chromosome selon un facteur de mutation. Ce facteur est la probabilité qu'une mutation soit effectuée sur un individu. Cet opérateur est l'application du principe de variation de la théorie de Darwin et permet, par la même occasion, d'éviter une convergence prématurée de l'algorithme vers un extremum local.[9]

Voici un exemple de mutation sur un individu ayant un seul chromosome :

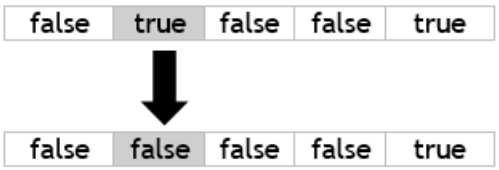

Figure 6.4 – Exemple de mutation

Avec ces trois opérateurs d'évolution, nous pouvons appliquer les algorithmes génétiques.

6.4 Les étapes des algorithmes génétiques

Les principes de bases étant expliqués, voici le fonctionnement des algorithmes génétiques :

- ***L'initialisation de la population*** est l'étape qui permet la création de la population initial, c'est le point de départ de notre algorithme.
- ***L'évaluation de la population*** consiste à calculer la fonction fitness pour chaque solution potentiel (bonne ou mauvaise).
- ***La sélection des meilleurs parents*** depuis la population actuel, on applique ***le croisement*** sur chaque paire des parents sélectionnés pour obtenir les nouveaux enfants. On réalise ensuite ***une mutation*** sur les enfants obtenues lors du croisement.
- Si on arrive à max itérations on sort de la boucle, l'algorithme retourne ***Sbest*** le meilleur résultat obtenu.
- Si le nombre max d'itération n'est pas encore atteint, on ***remplace la population actuel*** par une nouvelle population qui se compose des nouveaux enfants plus (+) les meilleurs parents (parents qui ont la plus grande fitness).

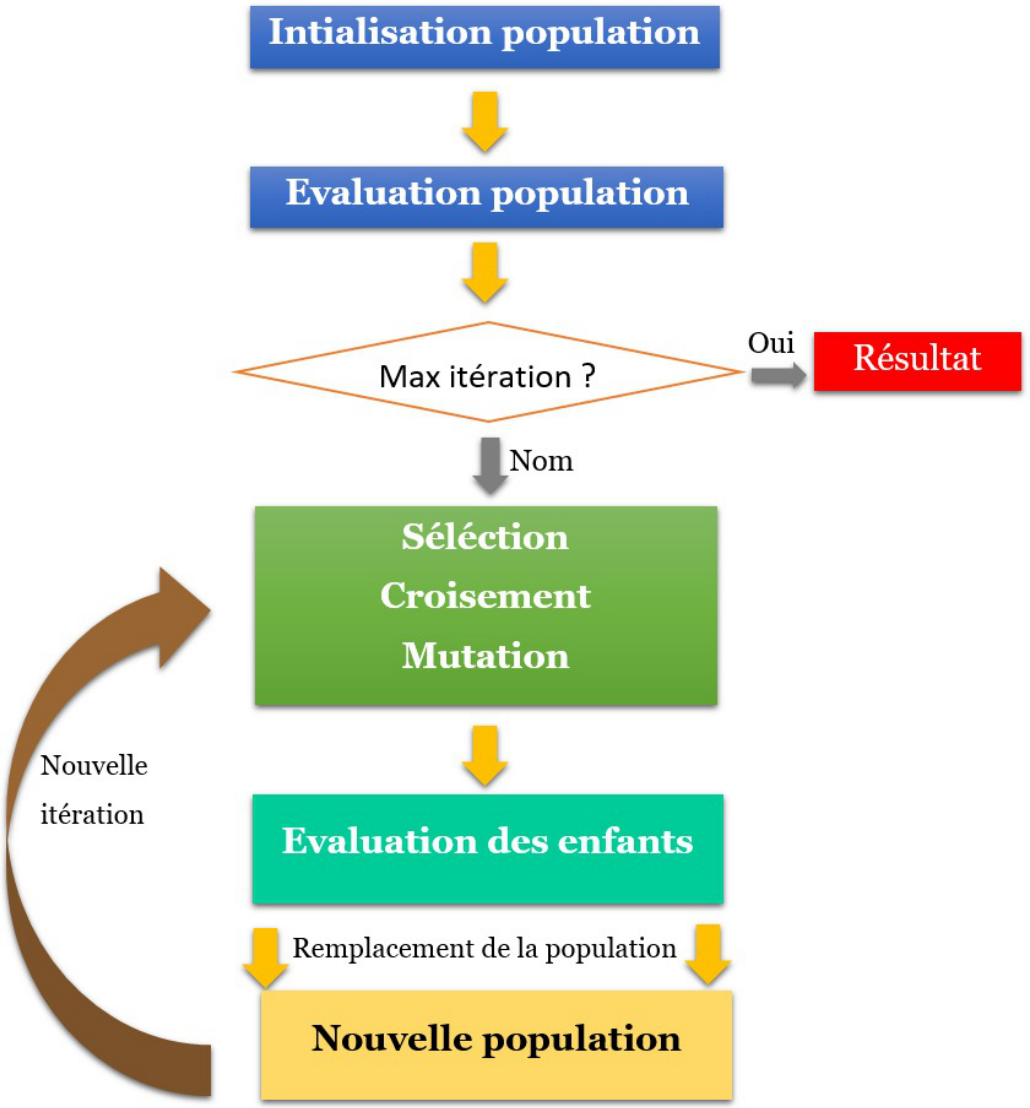

Figure 6.5 – Étapes des algorithmes génétiques

6.5 Pseudo-code

Dans le but de donner une meilleur description de fonctionnement des algorithmes génétiques nous allons présenter le pseudo code détaillé des algorithmes génétiques dans la figure ci-dessous.

Algorithmes génétiques

Entrée : M : Taille de population
variables ; *// liste de tous les variables*
variables.size() ; *//taille d'une solution*
clauses ; *//liste de tous les clauses*
//paramétres emiriques
max_iter : nbr des itérations
tc : taux de croisement
point_croisement
tm : taux de mutation
Sortie : SGbest *// meilleur solution global pour tous les itérations*

//génération aléatoire des solutions, une solution est un tableau de variable, donc on va gérérer M tableaux de variables aléatoirement.
Population=initialisation_population(population,variables,M);

//Evaluation de la population
//Calculer la fonction fitness pour chaque solution s de la population
Evaluation_population(population);

//Récupérer GSbest : la meilleur solution pour la population actuel (initial)
SGbest:= GetBestSolution(Population) ;

```
Tantque(Not max_iter) faire
    //Séléction des meilleurs parents pour le croisement (M/2) si M est paire
    //Selection des parents  en utilisant la méthode ranking(classement selon fitness)
    parents=SelectionParentsMethodRanking(population,M);
    //initialement aucun enfants
    enfants={null}
    //pour chaq paire p1,p2 de parents on va appliquer croisement et mutation
    pour p1,p2 ∃ parents faire
        enfant1=croisement(p1,p2,tc, point_croisement) ;
        enfant2=croisement(p2,p1,tc, point_croisement) ;
        enfant1=mutation(enfant1,tm) ;
        enfant2=mutation(enfant2,tm) ;
        //ajouter chaque enfant à l'ensemble de tous les enfants
        enfants.add(enfant1) ;  enfants.add(enfant2) :
    Fin pour
    //Evaluation des enfants
    Evaluation_population(enfants);
    //Récupérer SBest (l'enfant qui a la plus grand fitness)
    Sbest= GetBestSolution(enfants) ;
    //comparer GSbest avec SBest
    Si ( f(Sbest) > f(SGbest) ) alors
        GSbest=SBest;
    Fin si
    //Remplacement de la population
    //Remplacer par les nouveaux enfants plus les meilleurs  parents (qui ont la plus grande fitness)
    population=remplacement(parents,enfants ,M);
Fin Tantque
Retourner SGbest ;
```

Figure 6.6 – Pseudo-code des algorithmes génétiques

6.6 Analyse des résultats

Nous allons faire une analyse sur les résultats obtenus en appliquant les algorithmes génétiques pour résoudre le problème SAT.

Nous avons analysé la satisfaisabilité (nbr des clauses satisfaites) selon 4 degré de mutation (0,1 | 0.4 | 0.6 | 0.8) pour 10 fichiers choisies aléatoirement des deux benchmark UF-75 et UUF-75.

Benchmark UF-75

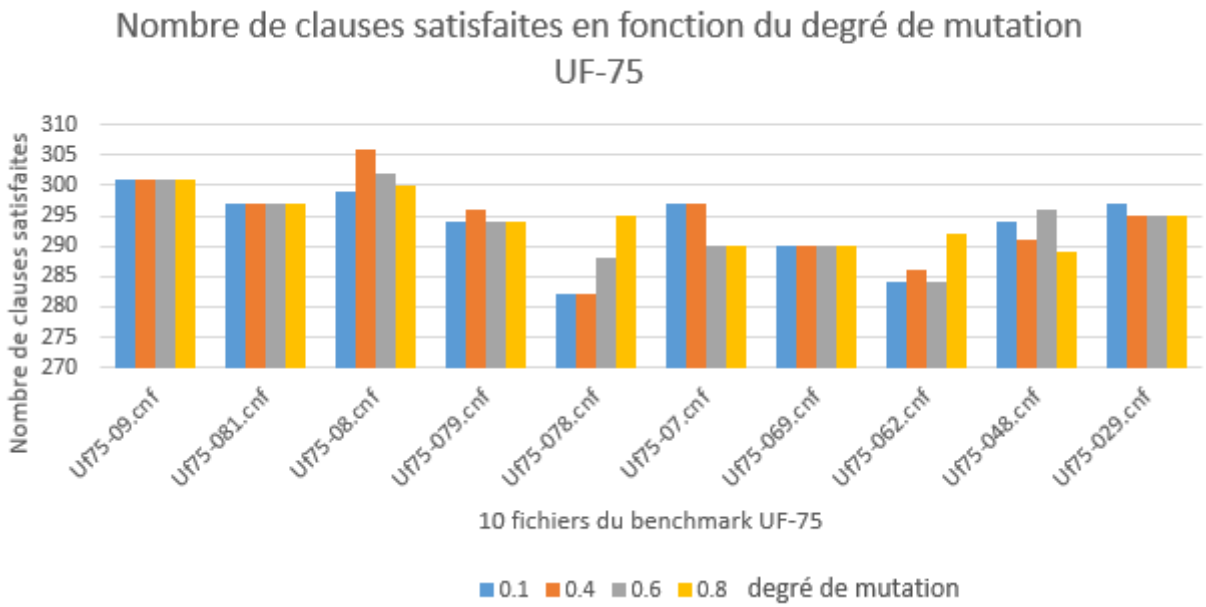

Figure 6.7 – Histogramme montrant les résultats de l'application des algorithmes génétiques sur 10 fichiers du Benchmarks UF-75 selon des degrés de mutation variables.

Benchmark UUF-75

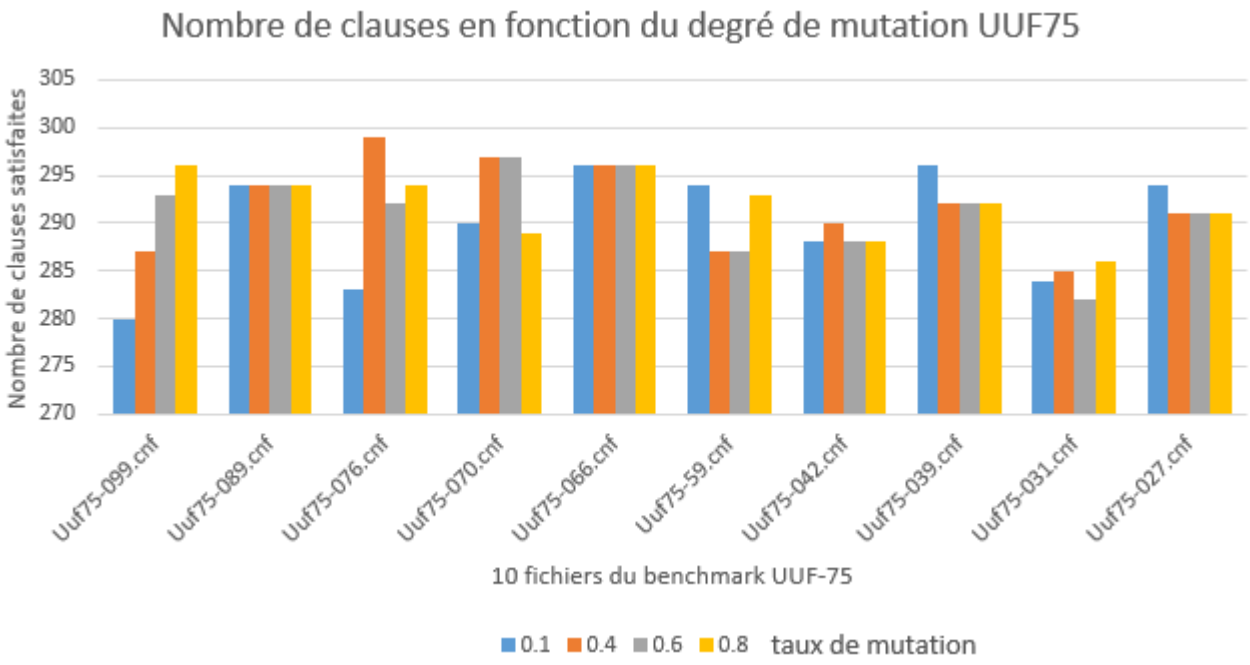

Figure 6.8 – Histogramme montrant les résultats de l'application des algorithmes génétiques sur 10 fichiers du Benchmarks UUF-75 selon des degrés de mutation variables.

Benchmark UF-75 : pour les fichiers : 09, 081, 08,079 et 07 nous constatons que le degré de mutation 0.4 permet de satisfaire le plus grand nombre de clauses.

Benchmark UUF-75 : pour les fichiers : 066, 059, 039 et 027 nous remarquons que le degré de mutation 0.1 donne le plus grand nombre de clauses satisfaites.

Pour les paramètres empiriques suivants :

Taille de population M=100.
Taux de mutation = 0.4.
Max itération = 100.
Taux de croisement = 1.

Nous obtenons le résultat suivant :

Le taux de statisfiabilité de 10 fichiers du benchmark UF 75 est 87,43 %.

Le taux de statisfiabilité de 10 fichiers benchmark UUF75 est de 80,85 %.

benchmark	Nom du fichier	Nombre de clauses satisfaites	pourcentage
UF-75	uf75-09.cnf	301	92,82 %
	uf75-081.cnf	297	91,38 %
	uf75-08.cnf	206	63,38 %
	uf75-079.cnf	296	91,08 %
	uf75-078.cnf	282	86,77 %
	uf75-07.cnf	297	91,38 %
	uf75-069.cnf	290	89,23 %
	uf75-062.cnf	286	88 %
	uf75-048.cnf	291	89,53 %
	uf75-029.cnf	295	90,76 %
	Total		87,43 %
UUF-75	uuf75-099.cnf	287	88,30 %
	uuf75-089.cnf	294	90,46 %
	uuf75-076.cnf	299	92 %
	uuf75-070.cnf	297	91,38 %
	uuf75-066.cnf	296	91,08 %
	uuf75-059.cnf	287	88,30 %
	uuf75-042.cnf	290	89,23 %
	uuf75-039.cnf	292	89,84 %
	uuf75-031.cnf	285	87,69 %
	uuf75-027.cnf	291	89,53 %
	Total		80,85 %

Table 6.1 – Tableau montrant le résultat de l'application des algorithmes génétiques sur 10 fichiers des benchmarks UF-75 et UUF-75.

6.7 Discussion des résultats

On constate que les taux de mutation 0.6 et 0.4 donnent les meilleurs résultats pour l'ensemble global des fichiers pour les deux benchmarks.

En appliquant les algorithmes génétiques, nous avons rencontré le problème de la convergence prématuré de sorte qu'après 100 itérations ou après 500 itérations la population ne s'améliore pas et SGbest reste la même. On est bloqué dans un état de stagnation.

L'application des algorithmes génétiques n'a pas donnée la solution optimale (satisfaire 325 clauses) mais une solution approchée dans un temps raisonnable. Cependant, pour les algorithmes génétiques nous avons obtenu une performance moins meilleure que l'algorithmes A*.

7. Partie 3 : Ant Colony System

7.1 L'intelligence en essaim

L'intelligence en essaim fait référence à une sorte de capacité de résolution de problèmes qui émerge dans les interactions de simples unités de traitement de l'information. Le concept d'un essaim suggère la multiplicité, la stochasticité, le caractère aléatoire et le désordre, et le concept d'intelligence suggère que la méthode de résolution de problèmes est en quelque sorte réussie.

Les unités de traitement de l'information qui composent un essaim peuvent être animées, mécaniques, informatiques ou mathématiques ; ils peuvent être des insectes, des oiseaux ou des êtres humains ; il peut s'agir d'éléments de réseau, de robots ou de postes de travail autonomes ; ils peuvent être réels ou imaginaires.[10]

Dans ce TP, nous nous intéréssons tout particulièrement à l'intelligence des colonies de fourmis. Nous allons appliquer l'algorithme de ACS (Ant Colony System) sur un problème de satisfiabilité pour parvenir à une solution qui va résoudre ce problème, nous allons étudier la performance de cette méthode, et comparer les résultats obtenus avec ceux des méthodes utilisées précédemment.

7.2 Ant Colony System

L'idée fondamentale des algorithmes de colonie de fourmis est inspirée par la façon dont les fourmis naturelles réussissent à trouver de la nourriture. Les fourmis communiquent via une substance appelée phéromone afin de trouver les chemins les plus courts du nid aux sources de nourriture. [10]

Sélection de la route

Chaque fourmi construit une solution en sélectionnant de manière itérative un nœud qu'elle n'a pas encore visité. À chaque étape de ce processus, les nœuds non encore sélectionnés forment l'ensemble candidat. Le nœud à ajouter à l'itinéraire actuel est choisi par rapport à la position actuelle de la fourmi, de manière probabiliste, dans l'ensemble candidat. [10]

La règle proportionnelle pseudo-aléatoire est utilisée pour calculer la décision, où sont utilisées deux variables :

— q : qui est une variable aléatoire uniformément distribuée dans [0,1].

— q_0 : est un paramètre empirique appartenant à [0,1].

Mise à jour locale de phéromone

Aussi appelée mise à jour en ligne qui se produit à chaque fois qu'une fourmi construit une solution, en changeant le niveau de phéromone dans état qu'elle traverse.

Le but de la mise à jour locale des phéromones est de s'assurer que les mêmes liens ne sont pas inclus encore et encore formant des circuits très similaires.[10]

Mise à jour globale de la phéromone

Aussi appelée mise à jour hors ligne. une fois que toutes les fourmis d'une même génération ont fini de construire leur solution, la phase de mise à jour globale de la phéromone suit. Seuls les routes inclus dans la meilleure solution globale reçoivent la phéromone. Le but de cette phase est de renforcer les routes qui appartiennent à la meilleure solution globale.[10]

7.3 Pseudo-code de ACS

La figure ci-dessous représente le pseudo-code de l'algorithme Ant Colony System :

```
Algorithme ACS
Début
    Initialisation : phéromone, best : initialisé aléatoirement;
    Pour (i=1) à Max_iter faire :
        Répéter :
            Pour (k=1) à Nbfourmi faire :
                Choisir le prochain état en appliquant la règle de transition;
                Effectuer la mise à jour en ligne de la phéromone;
            Fait ;
        Jusqu'à ce que chaque fourmi a construit sa solution;
        Mise à jour de bestSolution;
        Effectuer la mise à jour hors ligne de la phéromone;
    Fait ;
Fin
```

Figure 7.1 – Pseudo-code de l'algorithme ACS

7.3.1 Construction de la solution

Construction de la solution
Entrée : l : ensemble des états
Sortie : Sol : tableau de solutions

Début
Pour (i=1) à nb_lit faire :
Utiliser la règle de transition pour choisir le prochain état (littéral) ;
Supprimer l'état choisi de l'ensemble des états ;
Ajouter l'état sélectionné dans le tableau Sol ;
Fait ;
Retourner Sol ;
Fin

Figure 7.2 – Pseudo-code de la construction de la solution - ACS -

Chaque fourmi k construit sa propore solution avec un processus itératif, elle commence par appliquer la règle de transition sur les $2 * n$ littéraux dans le but de sélectionner un seul littéral pour commencer la solution.

Itérer le processus pour les $2 * (n - 2)$ littéraux restants jusqu'à la génération d'une solution avec n littéral.

7.3.2 Stockage de la phéromone

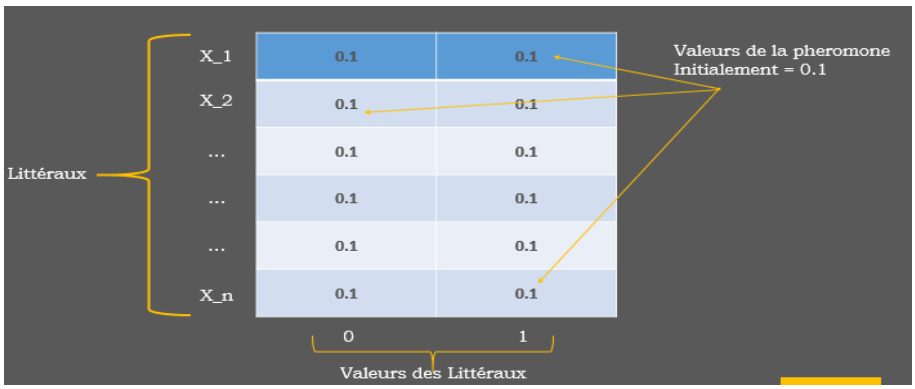

Figure 7.3 – stockage de la phéromone

— La phéromone est déposée sur chaque littéral de l'instance, elle représente l'information de l'utilisation précédente de ce littéral.

— Plus il y a de phéromone sur un littéral, plus augmente la probabilité que ce littéral soit choisit encore.

— L'information de la phéromone est implémentée en utilisant une matrice $n * 2$

— La matrice est initialisée avec une petite valeur 0.1

7.4 Analyse des résultats

L'algorithme de Ant Colony System a été très performant et a donné des solutions idéales. les remarques que nous pouvons faire à propos de cet algorithmes sont :

— Il a satisfait toutes les clauses de tous les fichiers des deux benchmarks UF-75 et UUF-75.

— Le temps moyen d'exécution de chaque fichier est inférieur à 6 secondes.

	Taux de satisfiabilité	Temps	Temps moyen
Uf75-325	100 %	10 minutes	6 secondes
UUf75-325	100 %	6.6 minutes	4 secondes

Table 7.1 – Temps écoulé - ACS -

7.5 Comparaison générale

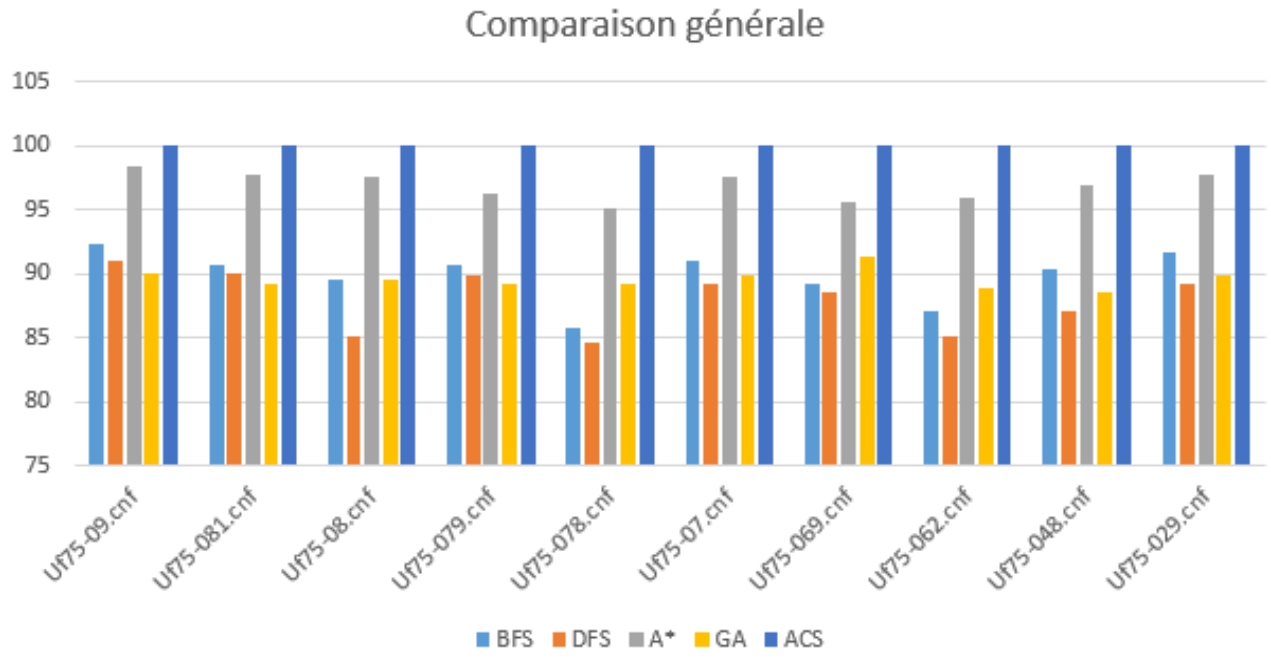

Figure 7.4 – Comparaison générale des algorithmes

Après analyse et observation des résultats des algorithmes de recherche aveugles (BFS et DFS), les algorithmes basés sur des heuristiques (A*), les algorithmes génétiques et ceux des colonies de fourmis, nous constatons clairement que les algorithmes de recherches aveugles sont les moins efficaces, ils prennent beaucoup de temps à trouver une solution ce qui n'est pas admissible. ensuite nous avons l'algorithme A* qui, malgré qu'il prend beaucoup de temps retourne un résultat plutôt satisfaisant avec un pourcentage de 96 % de satisfiabilité. En ce qui concerne les algorithmes génétiques, nous pouvons dire que ce sont des algorithmes efficaces qui produisent un résultat assez rapidement mais ne sont pas arrivés à trouver une solution qui satisfait 100 % des clauses. Enfin, l'algorithme de colonie de fourmie (ACS) s'est présenté comme l'algorithme le plus efficace en terme de temps et de qualité de résultat. en effet, ce dernier a satisfait 100 % des clauses en un temps minime.

Conclusion

Ce projet nous a permis d'appliquer différentes techniques dans le but de résoudre le problème de satisfaisabilité. SAT est considéré comme un problème de décision difficile NP-Complet car il n'existe aucun algorithme pour le résoudre dans un temps raisonnable. En effet, ce projet nous a confirmé l'explosion combinatoire qui se produit lorsque nous appliquons les méthodes exactes pour résoudre les problèmes de très grandes tailles. Ainsi les heuristiques peuvent aider à guider la recherche mais ils resteront insuffisantes pour arriver à la solution la plus optimale. Donc, il est nécessaire d'utiliser d'autres techniques plus performantes comme les métaheuristique.

La méta-heuristique bio-inspirée colonie de fourmis (ACO) a donnée les meilleurs résultats, la solution la plus optimale qui satisfait tous les instances du problème SAT. Nous concluons que les méta-heuristiques ACS (Ant Colony System) sont les plus efficaces pour résoudre le problème SAT.

Références

[1] URL : https://blog.pasithee.fr/2013/02/25/le-probleme-sat/ (visité le 11/01/2020).

[2] URL : https://fr.wikipedia.org/wiki/Probl%C3%A8me_SAT (visité le 11/01/2020).

[3] URL : https://people.sc.fsu.edu/~jburkardt/data/cnf/cnf.html (visité le 12/01/2020).

[4] URL : https://fr.wikipedia.org/wiki/Algorithme_de_parcours_en_largeur (visité le 28/01/2020).

[5] H. Zerkaoui Drias. *Algorithmique moderne analyse et complexité*. OPU, 2017.

[6] *Présentation Algorithme génétique Ilhem Daoudi*. URL : https://fr.slideshare.net/IlhemDaoudi/algorithme-gntique-69209124?from_action=save (visité le 30/01/2020).

[7] S. N. Sivanandam. *Introduction to Genetic Algorithms*. Springer, 2006.

[8] *Présentation Algorithme génétique Ilhem Daoudi*. URL : https://fr.wikipedia.org/wiki/Algorithme_g%C3%A9n%C3%A9tique (visité le 30/01/2020).

[9] URL : http://igm.univ-mlv.fr/~dr/XPOSE2013/tleroux_genetic_algorithm/fonctionnement.html (visité le 30/01/2020).

[10] Madalina Raschip Raluca Necula Mihaela Breaban. “Performance Evaluation of Ant Colony Systems for the Single-Depot Multiple Traveling Salesman Problem”. In : 9121 (2015), p. 12.