

République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
Université Benyoucef BENKHEDDA-Alger1



Faculté des Sciences
Département des Mathématiques et Informatique

Option : ISII

Projet

Implémentation des solutions pour le problème de satisfiabilité (SAT)

Réalisé par :

- BOURKIBA Ibrahim
- EL-MOUBAREK Fayçal

2021/2022

Table des matières

Partie I	3
1.1. Introduction.....	3
1.2. Traitement des fichiers.....	3
1.3. Structure de données générales	3
Partie II	4
2.1. Algorithme DFS “Depth First Search”	4
2.2. Algorithme A*	6
2.3. Algorithme Génétique	8
2.4. Système de colonies de fourmis (ACS)	10
3. Partie III	13
3.1. Etude Comparative	13
3.2. Conclusion	14

Partie I

1.1.Introduction

Le problème de Satisfiabilité SAT est un problème noté NP-Complet qui permet, grâce à de nombreuses méthodes et moyens, de vérifier si un ensemble de clauses est satisfiable ou pas, celui-ci est central en informatique et dans le domaine de l'intelligence artificielle.

Dans ce projet nous cherchons à appliquer des solutions de résolution sur ce problème grâce à des algorithmes exacts dont les méthodes aveugles et heuristiques pour but de constater l'explosion combinatoire en les implémentant en Java et ce grâce à une base de données fournies.

De ce fait nous exposons dans ce rapport l'analyse de ses solutions en 3 sections différentes les heuristiques, les algorithmes génétiques, et enfin la colonie de fourmi.

1.2.Traitement des fichiers

- Les fichiers de test sont caractérisés par : 100 instances (100 fichiers), 325 clauses ou chaque clause est de longueur 3 et 75 variables.
- Le premier dossier 'uf75-325' comporte les instances satisfaites, et le second 'uuf75-325' comporte les instances insatisfaites.
- Ensuite pour chaque instance nous avons déplacé grâce à la classe Fichier les lignes qui ne représentent pas les clauses et codifié également une matrice
- Après la lecture des fichiers il est nécessaire de trouver les solutions en utilisant les algorithmes

1.3.Structure de données générales

- Tableau clause : représente une clause de taille i où i est le nombre de littéraux chaque case est soit x (pour une valeur de littérale positive), $-x$ (pour une valeur négative).
- Tableau Clauses : c'est un ensemble de clauses.
- Tableau Solution : c'est un tableau qui représente une solution pour le problème SAT, sa taille est égale aux nombres de variables ; chaque case est soit 1 (pour une valeur de littérale positive), 0 (pour une valeur négative) et -1 pour une valeur vide.

Partie II

2.1.Algorithme DFS “Depth First Search”

Cette méthode consiste à élaborer un arbre en commençant par les nœuds les plus profonds avant de passer à des nœuds moins profonds et de revenir au sommet de l'arbre, ce qui signifie que cette méthode doit terminer la création des nœuds qui sont au bas de l'arbre et remonter au sommet et appliquer le même processus à l'autre côté de l'arbre.

Modélisation :

Structure de données additionnelle :

- **NodeDFS** : Un nœud qui représente les littéraux et leurs signes.
- **OpenDFS** : Une pile qui contient les nœuds qu'on doit visiter
- **BestSolutionDFS** : Un tableau pour stocker les meilleures solutions.

Implémentation :

Afin d'implémenter une solution pour le problème SAT, il est nécessaire de faire une étude simple du problème pour faciliter la tâche d'implémentation, basée sur la création des classes nécessaires et de leurs attributs et méthodes, et pour cela, 5 classes ont été choisies pour gérer les différentes informations et caractéristiques relatives aux problèmes SAT.

i) Classe « NodeDFS » : Cette classe représente les littéraux (Variable X) et elle est définie par deux variables « valeur : représente la valeur du littéral » et « signe : représente si la valeur du littéral est positive ou négative »

ii) Classe « ClausesSetDFS » : C'est la classe qui va extraire les clauses du fichier « .cnf » et les stocker dans un tableau. On aura comme variables dans cette classe :

- **clausesDFS** : c'est un tableau (ArrayList) de type « clauseDFS ».
- **clauseSizeDFS** : cette variable représente le nombre de littéraux dans une clause.
- **numberVariablesDFS** : cette variable représente le nombre de variables s'il y a plus de ou moins de 75 variables

iii) Classe « ClauseDFS » : Cette classe représente une clause en utilisant un tableau (ArrayList) des littéraux de type « integer » de la clause.

iv) Classe « SolutionDFS » : Le rôle de cette classe est de faire le traitement sur la solution et vérifier si cette solution est bonne ou mauvaise selon le nombre de clauses satisfiables, cela en utilisant un tableau (ArrayList) de type « integer » qui contient une solution possible.

v) **Classe « AlgorithmDFS »** : Le rôle de cette classe est de traiter l'algorithme DFS « Depth First Search » pour cela nous avons implémenté la méthode « **static** SolutionDFS DFS(**long** execTimeMillis,ClausesSetDFS clset) » Qui a comme entrée la durée de recherche et les clauses de notre fichier « .cnf », en sortie nous aurons « **return** bestSolutionDFS » un tableau de solution qui satisfait le maximum de clauses.

La modelisation de cette classe est comme suit :

- **OpenDFS** : Une pile de type « integer » elle contient les littéraux qui ne sont pas instanciés ($x = 0$)
- **SolutionDFS** : Tableau (ArrayList) de type « integer » son rôle est le stockage de la solution actuelle
- **BestSolutionDFS** : Tableau (ArrayList) de type « integer » son rôle est le stockage de la meilleure solution qui satisfait le maximum de clauses.

Analyse et Résultat :

• **Complexité temporelle :**

Après plusieurs tests sur différentes machines, nous avons constaté que l'algorithme de recherche en profondeur prend beaucoup de temps (Complexité exponentielle $O(2^n)$) pour arriver à satisfaire un nombre acceptable de clause.

• **Nombre de clauses satisfaites :**

Le nombre de clauses satisfaites ne dépasse pas les 290, un taux de satisfiabilité moyen égale à 87% pour chaque dossier dans une durée de 1000 secondes / 16.6 minutes.

Tableaux Des exécutions :

CS : Clauses satisfaites, **Temps** : temps d'exécutions **%** : pourcentage de satisfiabilité.

UF75.325.100				UUF75.325.100			
N° Fichier	CS	%	Temps (s)	N° Fichier	CS	%	Temps (s)
1	284	87.38	9.97	1	287	88.30	10.00
2	289	88.92	10.02	2	287	88.30	10.01
3	282	86.76	10.07	3	286	88.00	9.97
4	293	90.15	10.01	4	287	88.30	10.00
5	291	89.53	9.92	5	290	89.23	10.02
6	284	87.38	9.95	6	282	86.76	10.00
7	287	88.30	10.00	7	283	87.07	9.99
8	284	87.38	10.01	8	279	85.84	10.01
9	288	88.61	10.05	9	293	90.15	9.98
10	291	89.53	10.02	10	282	86.76	10.05

2.2.Algorithme A*

Dans cette partie, nous avons implémenté une solution pour le problème de satisfiabilité en suivant les étapes qu'on a vue afin de construire l'algorithme A*.

Vue que l'objectif de cet algorithme est de maximiser la fonction fitness alors :
 $f(x)=g(x)+h(x)$

H(x) : Heuristique est le nombre d'apparition x dans les différentes clauses

G(x) : le coût est le nombre de clauses qui seront satisfaites pas la solution traitée.

Modélisation :

Structure :

- **Liste de type 'Liste'** : Afin d'établir le chainage, chaque nœud contient une liste tel que tous les nœuds développés à partir de ce nœud y seront stockés.
- **Liste : ouvert** : contient le nœud à visiter.
- **Liste : ferme** : contient les nœuds déjà visités.

Implémentation :

Notre modélisation a conduit à la création des classes suivantes :

i) **Classe « Liste »** : Elle représente le type utilisé pour chaque nœud de l'arbre.

ii) **Classe « AEtoile »** : Elle comporte les méthodes suivantes :

- **VerifierExistence ()** : pour vérifier l'existence d'un nœud dans une liste
- **AjouterNoeud ()** : Afin d'ajouter un nœud, par ordre décroissant de la liste.
- **Solution_Racine ()** : elle retourne la solution depuis la racine.
- **Fonction_Fitness ()** : pour le calcul la valeur fitness de la solution.
- **A_EtoileFunction ()** : elle représente la méthode principale qui contient le corps de l'algorithme A*.

Analyse et Résultat :

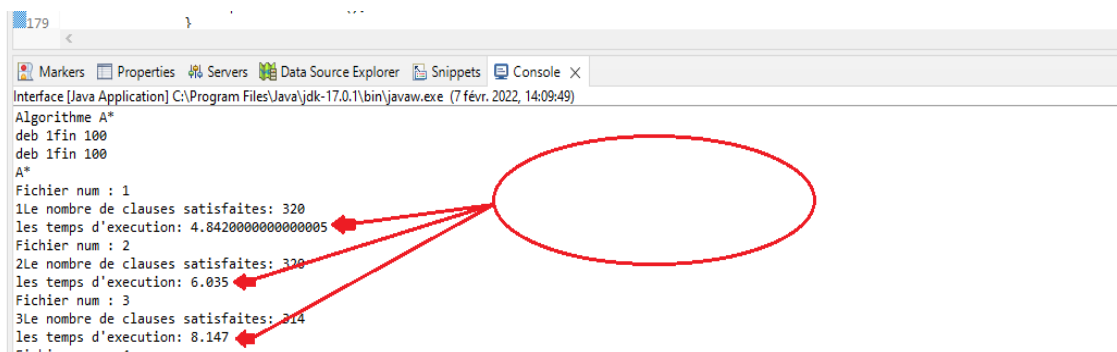
« La complexité temporelle et spatiale de l'algorithme A* est de $O(2^{75})$ »

L'algorithme prend beaucoup de temps lors de l'exécution plus de 5 heures pour chaque instance, ce qui est énorme. Aussi, l'algorithme stagne dans une phase de stabilité (optimum local), l'exécution continue sans avoir de meilleurs résultats

Donc pour minimiser le temps, nous avons fixé le nombre d'itérations durant la stabilisation à 1000 itérations, l'exécution des 100 fichiers a pris seulement 8.4 minutes

Après avoir choisi l'algorithme A* et sélectionner le dossier (UF75//UUF75).325.100 on définit le nombre de fichiers(instances) à traiter entre le 1er et le 100eme fichier.

- En exécutant l'ensemble de dossier le programme affichera le temps d'exécution final pour tous les fichiers sélectionnés dans le paramètre « Fichier ». Un exemple d'un affichage de temps d'exécution pour l'ensemble de 100 fichiers :603.46 secondes/8.4 mins
- En cliquant sur le bouton suivant dans la fenêtre contenant le temps d'exécution un tableau affichera les résultats des fichiers (Instances) sélectionnés.
- Afin d'avoir le temps d'exécution pour chaque fichier on les a affichés dans la console



```

Interface [Java Application] C:\Program Files\Java\jdk-17.0.1\bin\javaw.exe (7 févr. 2022, 14:09:49)
Algorithme A*
deb 1fin 100
deb 1fin 100
A*
Fichier num : 1
1Le nombre de clauses satisfaites: 320
les temps d'execution: 4.8420000000000005
Fichier num : 2
2Le nombre de clauses satisfaites: 320
les temps d'execution: 6.035
Fichier num : 3
3Le nombre de clauses satisfaites: 314
les temps d'execution: 8.147

```

Tableaux Des exécutions :

CS : Clauses satisfaites, Temps : temps d'exécutions %: pourcentage de satisfiabilité.

UF75.325.100				UUF75.325.100			
N° Fichier	CS	%	Temps (s)	N° Fichier	CS	%	Temps (s)
1	320	98%	4.82	1	319	98%	7.72
2	320	98%	6.03	2	312	96%	8.27
3	314	97%	8.14	3	317	97%	5.09
4	317	98%	5.46	4	318	97%	2.26
5	319	98%	8.51	5	315	96%	7.96
6	317	98%	8.50	6	317	97%	3.44
7	317	98%	5.35	7	314	96%	8.26
8	320	98%	7.80	8	321	98%	4.77
9	317	98%	8.94	9	318	97%	6.37
10	323	99%	5.36	10	316	97%	5.55

2.3.Algorithme Génétique

Dans cette partie, nous implémentons pour le problème de satisfiabilité un algorithme génétique il faut savoir que les algorithmes génétiques sont des Meta heuristique sans mémoire avec une recherche aléatoire qui consiste à faire évoluer une population.

Modélisation :

- Tableau Individu : représente une solution contenant un ensemble de variable autrement dit un chromosome avec des gènes.
- Les individus ont des pourcentages selon l'intervalle auquel ils appartiennent.
- Nous avons utilisé pour la sélection la méthode de la roulette, pour la mutation le bitflip et pour le croisement celle du point unique.
- L'utilisation d'un taux de croisement et de mutation est pris en compte en étant un pourcentage.
- L'évaluation des individus s'est faite grâce à la fonction **getfitness()**.

Implémentation :

i) **Classe Recherche** : Cette classe consiste tout simplement à traiter l'algorithme génétique grâce à deux fonctions implémenter a l'intérieure :

La première est Genetic Algorithm c'est la fonction principale de notre traitement elle nécessite en entrée l'ensemble des paramètres nécessaires dont le nombre d'itération, le taux de croisement, le taux de mutation ainsi qu'un set de clauses, cette fonction donne en sortie une solution représenté sous forme de liste composer d'un ensemble d'individu, elle permet également d'effectuer toute les étapes de sélection, mutation, croisement , d'évaluation (grâce à la fonction fitness) et d'insertion nécessaire.

Pour utiliser les clauses de nos fichiers nous avons opté à la création d'une classe **Clauses** : cette classe comporte un set de clauses récupérer des fichiers. cnf mis sous formes de tableau afin d'être manipuler.

La deuxième fonction est Cross c'est la fonction chargée du croisement entre les deux individus parents formant un individu enfant, elle récupère les clauses de notre set.

ii) Classe Individual : Cette classe représente un individu de la population (un chromosome), cet individu comprend des gènes.

On traite principalement dans cette classe l'évaluation de la solution grâce à la fonction fitness.

L'utilisation de l'individu se retrouve notamment dans la classe recherche.

iii) Classe Solution : Cette classe permet la représentation de la solution et sa manipulation, elle comprend :

La fonction **isSolution** et **SatisfiedClauses** sont déjà expliqués plus haut dans la partie recherche en profondeur.

La fonction **InvertLiteral** nécessaire pour la récupération des positions des gènes et l'utiliser lors de la mutation et effectuer le bitflip.

Analyse et Résultat :

Après avoir testé l'algorithme sur l'ensemble des données en initialisant les paramètres nécessaires à savoir le taux de croisement, et de mutation exclusive à l'algorithme génétique ainsi que la taille de la population et le nombre d'itération, voici un tableau qui résume les tests sur un des fichiers fournis :

100 fichiers pris en compte	Nombre d'itération	Taux de croisement	Taux de mutation	Taux Moyen de satisfiabilité	Le temps moyen d'exécution
1	20	30	20	91.88%	0,88 s
2	50	40	30	92.47%	0,90 s
3	100	50	40	92.61%	1 s

En conclusion l'algorithme génétique montre une efficacité considérable et donne de meilleurs résultats comparés aux autres méthodes, le fait de prendre 100 fichiers à chaque fois a pu légèrement impacter le temps d'exécution, sans cela en considérant 5 ou 6 fichiers le temps reste à 0 seconde.

2.4.Système de colonies de fourmis (ACS)

Le système de colonies de fourmis est une des métaheuristiques bio-inspirées les plus performants et qui est adaptée parfaitement aux problèmes modélisés par des graphes comme le problème du voyageur du commerce, elle permet de lancer plusieurs agents (fourmis) à partir de différents points de départ mais orientés tous vers le même objectif à savoir la construction de la meilleure solution, guidé par des règles des transitions qui sont à leur tour générées par le calcul de la fonction objective et du taux de phéromone, et cela pour chaque itération d'une solution potentielle (bonne ou mauvaise).

Modélisation :

Problème :

Clauses :

- $C = \{c_1, c_2, \dots, c_{325}\}$ ensembles de clauses ou chaque 'ci' est une disjonction de littéraux.
- Nombre de clauses : 325
- Taille d'une clause : 3

Variables :

- $X = \{1, 2, 3, \dots, 75\}$ ensemble de variables booléennes.
- Nombre de variables : 75
- Taille d'une variable : soit 1 (ex : 6, 9), soit 2 (ex : 17, 44)

Littéraux :

- Un littéral est une variable booléenne avec ou sans l'opérateur de négation '-'.
- Nombre de littéraux : $75 * 2 = 150$
- Taille d'un littéral : soit 1 (ex : 8, 5), soit 2 (ex : 17, 44), soit 3 (ex : -58, -30)

Solution :

Solution bonne ou mauvaise : Un ensemble de 75 variables instanciés, chaque variable est soit sous sa forme positive ou négative.

Fonction objective : Fonction fitness qui représente le gain, qui est le nombre de clauses satisfaisantes : $0 \leq \# \text{clauses satisfaites} \leq 325$

Phéromone :

- Substance chimique sécrété par les glandes abdominales de la fourmi
- Une valeur calculée qui, comme la fonction fitness, nous aide à évaluer une solution.
- Il existe deux façons d'évaluer une solution avec la phéromone (online update, offline update).

Implémentation :

Problème :

Classe Literal : C'est un objet java caractérisé par une valeur de variable et sa forme (positive/négative).

Classe Clause : C'est un objet java présenté par une liste de littéraux : List<Literal> literals.

Classe Sat : Objet java qui est un ensemble de clauses : List<Clause> clauses.

Classe Parameter : Objet java qui définit les paramètres empiriques expérimentales :

```
static double ALPHA;  
static double BETA;  
static double EVAPORATION_RATE;  
static double INIT_PHEROMON;  
static int MAX_ITERATIONS;  
static int ANT_NBR;  
static double Q_0;
```

Solution :

Classe Solution :

- Objet java qui définit une solution potentielle.
- La solution de départ de la construction d'une solution par une fourmis est initialisée par l'algorithme A*.

Structure de données utilisée :

- Une solution est un vecteur de type integer et de taille : #variables.
- L'indice de chaque case 'i' + 1 = la variable 'i' (car le vecteur est 0-indexed)

Classe Display : Objet java qui nous permet de récupérer le temps du déroulement de l'algorithme ainsi que la solution 'best' avec la meilleure valeur de fitness.

Classe Ant : Objet java qui construit une solution et calcul les règles de transitions.

Classe ACS : C'est la classe qui décrit le déroulement de l'algorithme vu en cours.

Structure de données utilisée :

```
FITNESS = new int[Sat.NBR_LITERALS][2];
```

```
pheromon = new double[Sat.NBR_LITERALS][2];
```

La structure de données matrice représente parfaitement les valeurs de fitness et de phéromone, en effet, le nombre de ligne = nbr_littéraux = 75, et le nombre de colonnes = 2 (pour 0 et 1).

Pour l'évaluation de la fonction fitness pour chaque solution générées par les fourmis, nous avons stocker ces valeurs dans une Pile, et récupérer à la fin le max des fitness en utilisant la méthode de classe suivante: `Collections.max(stack)`

Analyse des résultats :

Nous avons fixé les paramètres empiriques suivants pour observer le nombre de clauses satisfaites CS et le temps d'exécution :

Alpha	Beta	Evap-rate	Init-phero	Q0	Ant nb	Itér
0.5	0.2	0.7	0.001	0.3	20	100

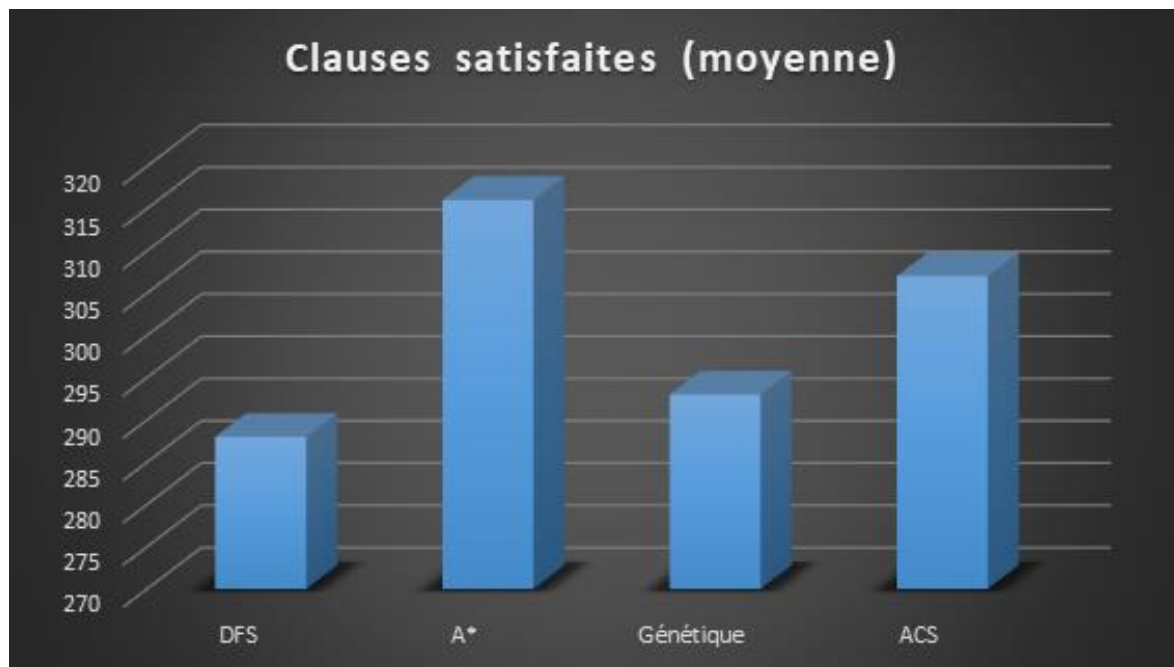
UF75.325.100		
N° Fichier	CS	Temps (ms)
1	307	59
2	309	144
3	310	157
4	311	224
5	310	253
6	310	381
7	309	400
8	310	480
9	308	516
10	311	536

Partie III

3.1.Etude Comparative

Durant ce projet nous avons traité le problème de satisfiabilité avec 4 algorithmes différents : d'une part l'Algorithme A* et DFS étant des recherches aveugles et d'une autre part l'Algorithme génétique et ACO étant des algorithmes évolutionnaires utilisant les meta heuristique.

Après l'exécution de chacun de ses algorithmes nous avons pu les représenter dans un graphe ci-dessous, on peut remarquer qu'aucun algorithme n'a atteint la satisfiabilité totale à savoir 100% de clauses satisfaites :



Selon le graphique , DFS a réussi à satisfaire en moyenne 288 clauses avec le max de clauses satisfaites 301 pour certaines instances en revanche le A* en satisfait largement plus et est en tête du classement avec en moyenne 316 clauses satisfaites avec un maximum de 323 clauses pour certaines instances .

L'AG quant à lui a satisfait en moyenne 293 instances et le dépasse le ACS avec 315 instances satisfaites.

Conclusion ACS et A* sont meilleurs dans le taux de clauses satisfaites .

En ce qui concerne le temps d'exécution de chaque algorithme voici la représentation :

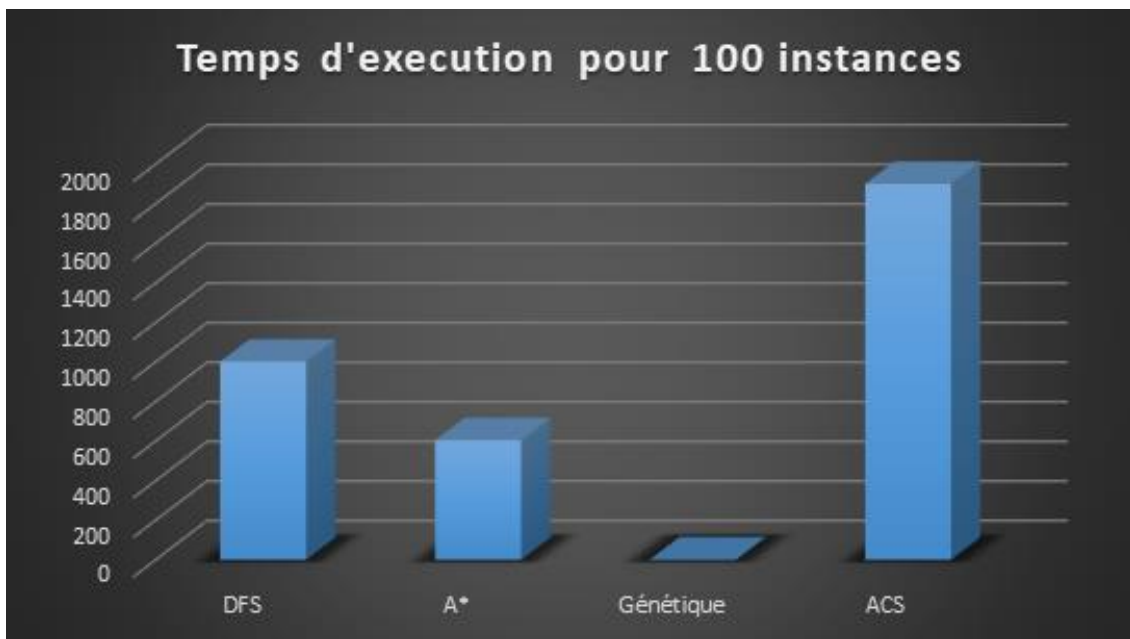
L'algorithme A* ici prend 8,4 minutes il faut souligner pour son cas que du à l'explosion combinatoire on a dû limiter le nombre d'itération en passant de 5 heures par instance à

8,4minutes, il reste néanmoins malgré son taux de satisfiabilité impuissant face à de grands espaces de recherche.

Le DFS en revanche effectue 1001,6 secondes soit 16,6 minutes.

AG dans ce volet est imbattable avec un maximum d'une seconde pour 100 instances à la fois.

ACS doit son efficacité au fait qu'il démarre d'une solution aléatoire générée par A* qui est déjà performant. Mais il prend beaucoup de temps par rapport aux autres algorithmes, 1h10 pour 100 instances



3.2.Conclusion

Le problème SAT est un problème très connu qui nécessite un certain nombre de solutions et dépend fortement de sa taille, il nécessite un ensemble de techniques qui permettent de trouver la meilleure solution.

Nous avons avec cette recherche mis en avant des algorithmes qui résolvent le problème SAT en imposant un ensemble de paramètres.

Même si les autres algorithmes étaient largement plus rapides que le A*, mais il a réussi à atteindre le meilleur taux de clauses satisfaites. (Sachant qu'on s'arrête toujours au 1^{er} optimum local)