République Algérienne Démocratique et populaire
Université des Sciences et de la Technologie Houari Boumediene

Faculté d'Electronique et Informatique
Département Informatique

Module : Métaheurstiques

Les métaheuristiques pour la résolution du problème de satisfiabilité SAT

Réalisé par :
MOUSSAOUI Amina
CHENNIT Naila

M1 SII G01
2017/2018

Sommaire

Partie I

Utilisation des méthodes aveugles et heuristiques pour la résolution du problème de satisfiabilité SAT

1.Introduction:

a. Objectifs:

En informatique implémenter l'approche basé sur l'espace des états représente un grand un important domaine de recherche étant données la quantité de problème dont on dispose, l'un des plus connu étant le problème SAT.

Le problème de satisfiabilité appelé aussi SAT est le premier problème qui a été montré NP-complet grâce au théorème de Cook en 1971. Ce dernier nous permet, étant donné une formule écrite sous forme normale conjonctive, de déterminer si elle est satisfiable ou non ; autrement dit si selon un ensemble de valeurs de variables , les clauses de la formule SAT sont-elle toutes vraies.
L'étude de ce problème et de ses variantes a permis de pointer sur la frontière entre les problèmes faciles et difficiles. En effet, SAT a aidé à classer beaucoup d'autres problèmes, par réduction polynomiale. De nos jours ce système de résolution est utilisé dans bon nombre de domaines tel que :
la cryptanalyse, la planification, les diagnostics, la vérification de matériels et de logiciels…etc.

Dans cette première partie, nous décrirons et analyserons les différentes méthodes de résolution du problème SAT à travers quatre algorithmes de recherche distincts :

- Profondeur d'abords
- Largeurs d'abords
- Cout uniforme
- A*

b. Méthodes de recherche:

1/ Recherche en largeur d'abord (BFS: *Breadth-first search*)

La recherche en largeur d'abord consiste à exploiter toutes les assignations des valeurs d'un sous ensemble de variables couche par couche et itérer le processus jusqu'à couvrir tout l'ensemble des variables et atteindre un certain niveau.
La numérotation associés à chaque nœud dans la figure 1.1 définie l'ordre de développement des nœuds dans l'arbre.

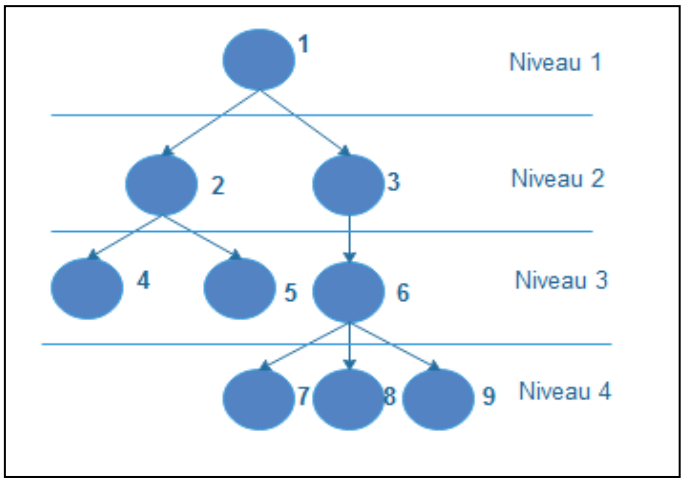


Figure1.1 Recherche en largeur d'abord

2/ Recherche en profondeur d'abord (DFS: Depth-first search):

La recherche en profondeur d'abord tente d'atteindre le but le plus vite possible en explorant immédiatement les successeurs de tous nœuds générés.

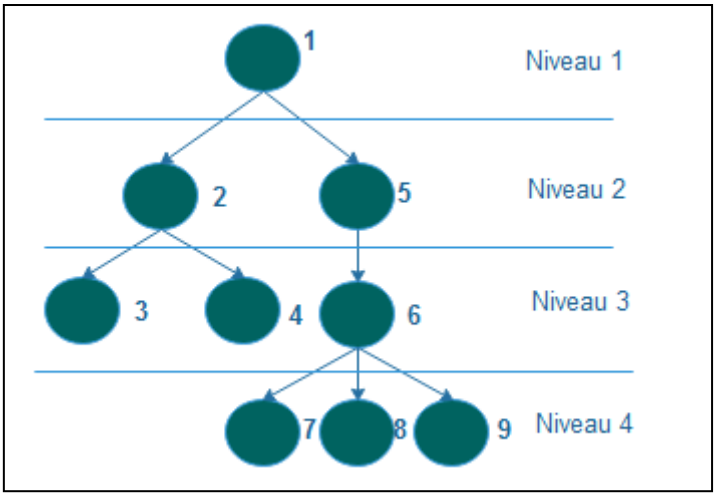


Figure1.2 Recherche en profondeur d'abords (ordre de développement de nœuds)

3/ Recherche cout uniforme:

Est une recherche aveugle tel que DFS et BFS, sauf que cette dernière est choisi le successeurs d'après la valeurs de la fonction.

4/ Recherche avec A*:

Contrairement aux trois méthodes de recherche vue plus haut, A* est une méthode informée. En effet cette dernière œuvre selon une fonction basé sur une heuristique et cela afin d'estimer le meilleur chemin.

2. Implémentation:

a. Structures de données

Input: fichier Benchmarks**.**

Traitement:

- Une matrice de clauses de [325][3] (325 clauses composées de 3 littéraux), obtenue à partir du fichier Benchmrks.
- Vecteur d'entier de taille [75] représentant les valeurs des littéraux (l'index représente le littéral).
- Vecteur d'entier de taille [75] représentant la fréquence d'apparition de chaque littéral dans la matrice de clauses (pour A*).
- Vecteur d'entier de taille [325] représentant les valeurs des clauses.
- Type nœud { père , fils_gauche , fils_droit ,profondeur, fonction}
- Liste open contenant les nœuds en cours de développement:
    1. Manipulée en tant que File (FIFO) pour le parcours en largeur d'abord.
    2. Manipulée en tant que Pile (LIFO) pour le parcours en profondeur d'abord.
    3. Manipulé en tant que liste triée pour le cout uniforme.
- Arbre de type Nœud
  Nous avons implémenté l'algorithme BFS en utilisant une structure statique de l'arbre, contrairement aux algorithmes DFS , A* et cout uniforme ou nous avons implémenté une structure dynamique .

Output: nombre de clauses satisfiables et taux de satisfiabilité.

b. Algorithme:

1/ DFS:

```
Procedure ProfondeurDabord( )
 Entrée fichier : matrice de littéraux [325][3]
 Variables :
 Open : Pile de nœuds
 root, temp,fils1,fils2 : nœuds

 Debut
        root.pere=null; root.profondeur=0;
         empiler(root,open);

  Tant que (Open ≠Ø) faire

          temp=depiler(Open);

           Si (Satisfiabile(temp,fichier)=325) alors Exit ;
            Sinon
             Si(temp.profondeur<75)
                 fils1.pere=temp;
                 fils1.profondeur=fils1.profondeur+1;
                 fils1.val=0;

                 fils2.pere=temp;
                 fils2.profondeur=fils2.profondeur+1;
                 fils2.val=1;

                 Empiler(fils1,Open);
                 Empiler(fils2,Open);
             Fsi;
           Fsi;
    FinTantque;
 Fin;
```

Illustration du parcours avec 2 variables:
La figure suivante représente le déroulement de la recherche avec DFS pour l'ensemble de clauses suivant:

C1=A+B
C2= ¬ A+B
C3= ¬A+¬B

La valeur de l'étoile représente le nombre de clauses satisfiables pour chaque chemin (en rouge) emprunté

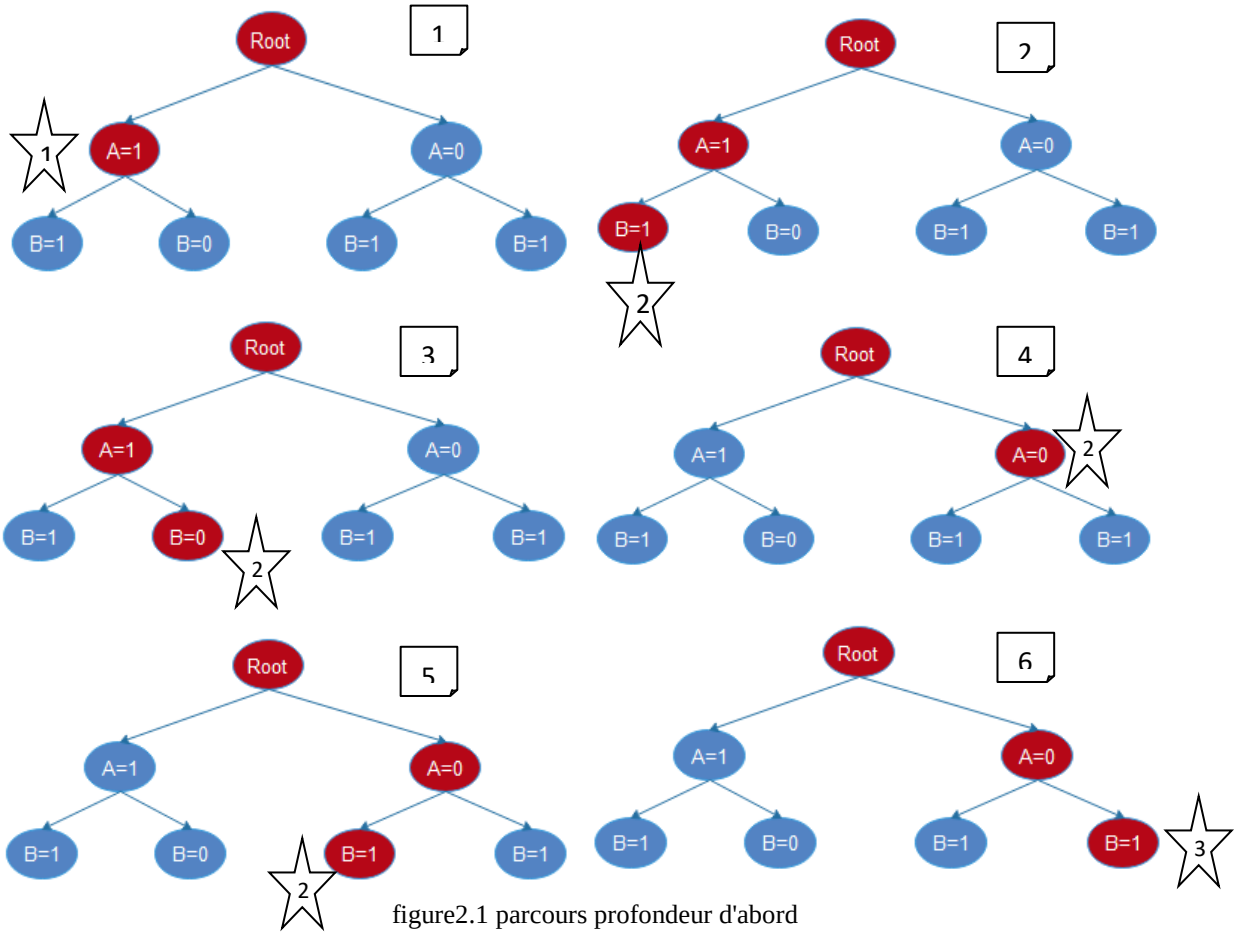


figure2.1 parcours profondeur d'abord

.
La condition d'arrêt est que le nombre de clause satisfiable soit égal au nombre de clauses
Pour cette exemple les valeurs qui satisfirent l'ensemble de clause fut trouvé au bout de la dernière itération.

2/ BFS:

```
Procedure largeurDabord()
    Entrée fichier : matrice de littéreaux [325][3]
    Variables : Open : File de noeuds
                root, temp,fils1,fils2 : noeuds

    Debut

        enfiler(root,open);

        Tant que (Open ≠Ø) faire

            temp=Défiler(Open);

            Si (Satisfiabile(temp,fichier)=325) alors Exit ;
              Sinon
                Si(temp.profondeur<75)
                    fils1.pere=temp;
                    fils1.profondeur=fils1.profondeur+1;
                    fils1.val=0;

                    fils2.pere=temp;
                    fils2.profondeur=fils2.profondeur+1;
                    fils2.val=1;

                    Enfiler(fils1,Open);
                    Enfiler(fils2,Open);
                Fsi;
              Fsi;

        FinTantque;

    Fin;
```

Illustration du parcours avec 2 variables:
La figure suivante représente le déroulement de la recherche avec BFS pour l'ensemble de clauses suivant:

C1=A+B
C2= ¬ A+B
C3= ¬A+¬B

La valeur de l'étoile représente le nombre de clauses satisfiables.

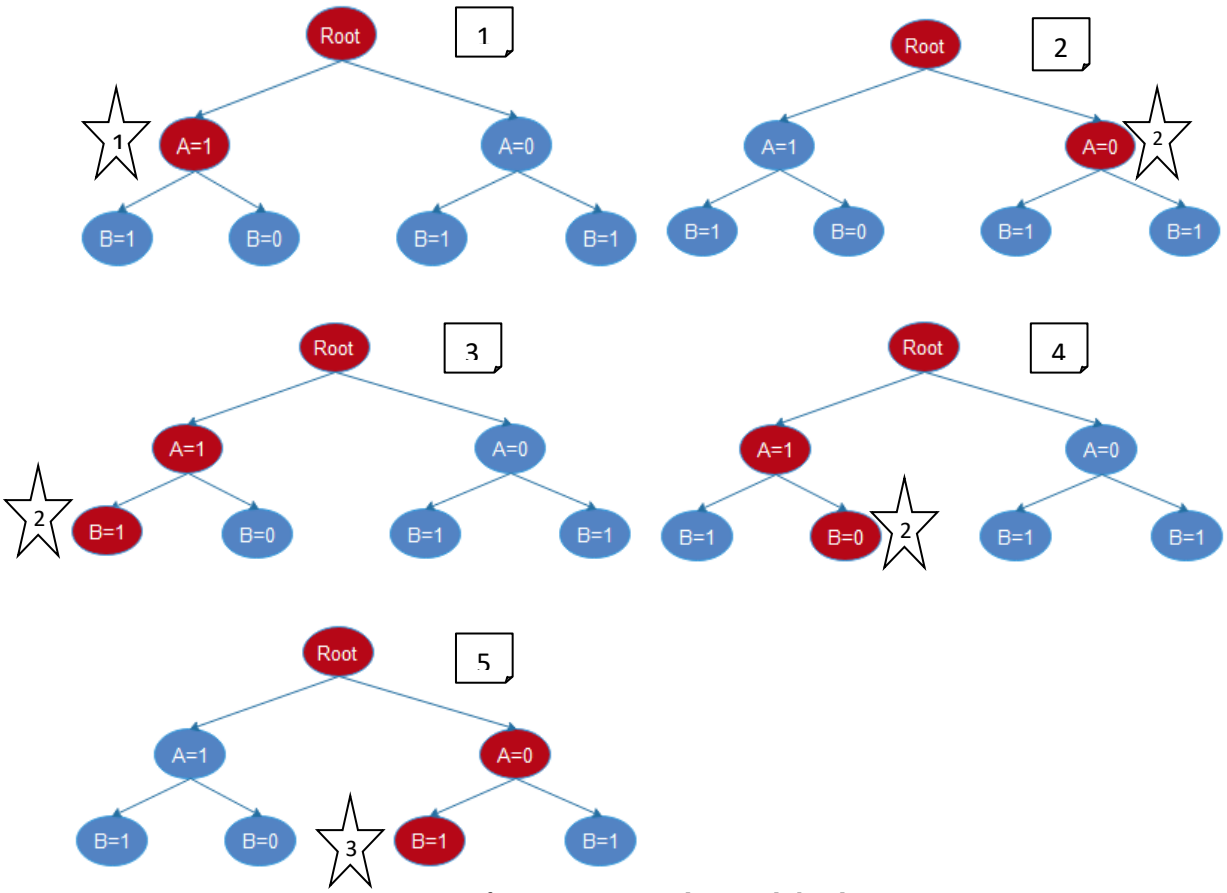


figure2.2 parcours largeur d'abord

Dans ce parcours, la condition d'arrêt est vérifiée au bout de la cinquième itération.

3/ Cout uniforme:

Procedure CoutUniforme()
Entrée : fichier : matrice de littéraux [325][3]

Variables:
feuilles: Array de noeuds
root, temp,fils1,fils2 : noeuds
nb_clause_satisfiable : entier

Debut

```
        root.pere=null; root.profondeur=0; root.fonction=0;

       temp=root;

  Tant que (temp.profondeur<75) faire

            fils1.pere=temp;
             fils1.profondeur=fils1.profondeur+1;
            fils1.val=0;
            nb_clause_satisfiable=satisfiable(fils1);
             fils1.fonction=nb_clause_satisfiable;
            ajouter fils1 à feuilles;


            fils2.pere=temp;
            fils2.profondeur=fils2.profondeur+1;
            fils2.val=1;
            fils2.fonction=nb_clause_satisfiable;
            ajouter fils2 à feuilles;

              trier(feuilles); //trie la liste par rapport a la fonction
              temp=getFirst (feuilles);


   FinTantque;

Fin;
```

4/ A* :

l'algorithme d'A* est identique à celui du cout uniforme excepté le fait qu'on rajoutera la valeur de l'heuristique à la valeur de la fonction de chaque nœud.
comme heuristique, nous avons opté pour la fréquence d'apparition du littéral dans le fichier testé.

3.Expérimentations:

a. Données:

-Fichier Benchmarks.

***Description**:*
- 20 fichiers en format CNF (20 instances).(10 satisfiables + 10 non satisfiables).
- Chaque fichier contient 325 clauses et 75 variables.
- Chaque clause est composée de 3 variables (SAT = 3).

b. Environnement de travail:

.

RAM	***8GO***
Processeur	*(R) Core™ i7-3210M CPU @ 2.50 GHz.*
Système d'exploitation	*Windows 10-64 bits*
Langage de programmation	*JAVA*
Environnement de développement	*Eclipse*

c. Interface:

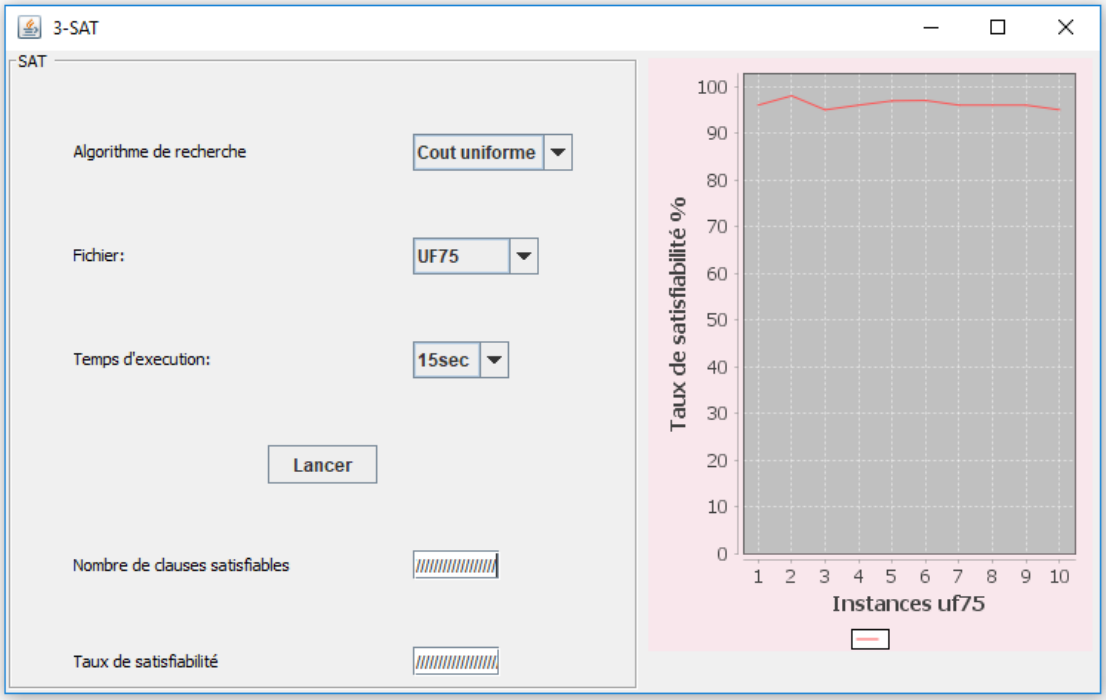


Figure4.1 Interface

L'instance sélectionné est "UF75 " . Donc l'exécution est faite pour les 10 instances de UF75 selon une duré de test fixé par l'utilisateur, cela est traduit par le graphe.

d. Résultats expérimentaux:

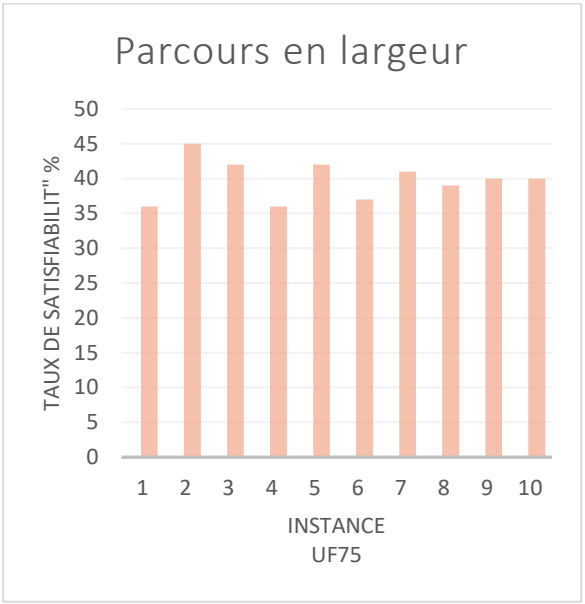
1/ Largeur d'abord :

Instances uf75:

Nom du fichier	Nombre de clauses satif Moyen **sur 10 exécutions**	Nombre de clauses satif minimal	Nombre de clauses satif maximal	Taux de satisfiabilité moyen
uf75-01.cnf	116	110	123	36.25%
uf75-02.cnf	148	127	169	45.53%
uf75-03.cnf	137	122	152	42.15%
uf75-04.cnf	117	114	121	36.00%
uf75-05.cnf	137	130	144	42.15%
uf75-06.cnf	122	121	128	37.53%
uf75-07.cnf	134	127	141	41.23%
uf75-08.cnf	127	113	141	39.07%
uf75-09.cnf	131	128	146	40.30%
uf75-010.cnf	132	121	143	40.61%

Instances uuf75:

Nom du fichier	Nombre de clauses satif Moyen **sur 10 exécutions**	Nombre de clauses satif minimal	Nombre de clauses satif maximal	Taux de satisfiabilité moyen
uuf75-01.cnf	120	135	105	36.92%
uuf75-02.cnf	147	140	152	45.23%
uuf75-03.cnf	132	128	150	40.61%
uuf75-04.cnf	131	126	142	40.30%
uuf75-05.cnf	132	125	143	40.61%
uuf75-06.cnf	134	132	136	41.23%
uuf75-07.cnf	129	125	136	39.69%
uuf75-08.cnf	131	130	145	40.30%
uuf75-09.cnf	147	142	152	45.23%
uuf75-010.cnf	129	126	141	39.69%



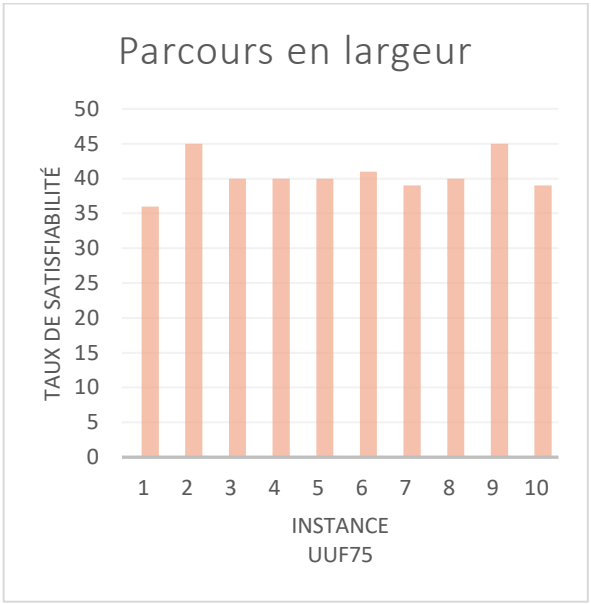


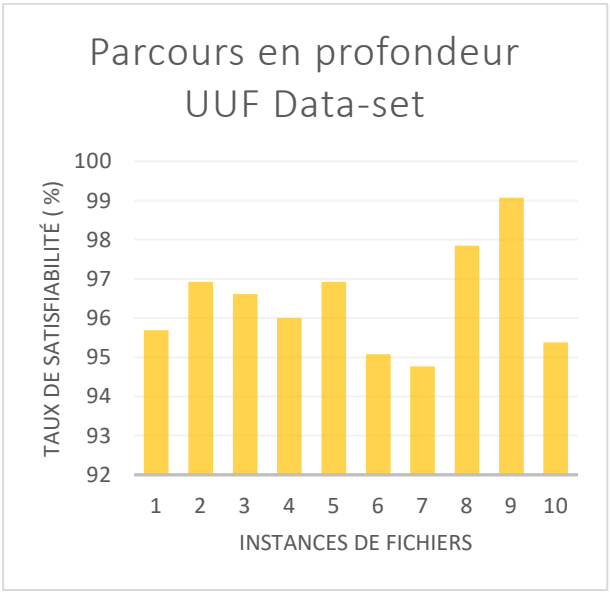
Figure3.1 taux satisfiabilité pour fichier uf75 et uuf75 parcours en largeur

Les résultats expérimentaux ont démontré l'inefficacité du parcours en largeur . Le taux maximal de clauses satisfiables avoisine les 50%.

2/ Profondeur d'abord :

Nom du fichier	Nombre de clauses satif Moyen **sur 10 exécutions**	Nombre de clauses satif minimal	Nombre de clauses satif maximal	Taux de satisfiabilité moyen
uf75-01.cnf	290	285	301	89.23%
uf75-02.cnf	301	298	308	94.06%
uf75-03.cnf	298	290	306	91.69%
uf75-04.cnf	303	298	307	93.23%
uf75-05.cnf	302	300	306	92.92%
uf75-06.cnf	299	290	303	92.00%
uf75-07.cnf	296	293	301	91.07%
uf75-08.cnf	290	284	295	89.23%
uf75-09.cnf	298	288	299	91.65%
uf75-010.cnf	301	291	302	94.06%

Nom du fichier	Nombre de clauses satif Moyen **sur 10 exécutions**	Nombre de clauses satif minimal	Nombre de clauses satif maximal	Taux de satisfiabilité moyen
uuf75-01.cnf	298	291	302	91,69%
uuf75-02.cnf	303	301	309	93,23%
uuf75-03.cnf	299	296	304	92.00%
uuf75-04.cnf	295	290	300	90,77%
uuf75-05.cnf	305	301	309	93,85%
uuf75-06.cnf	293	291	305	90,15%
uuf75-07.cnf	293	289	299	90,15%
uuf75-08.cnf	296	294	301	91,08%
uuf75-09.cnf	304	301	308	93,54%
uuf75-010.cnf	303	300	306	93,23%



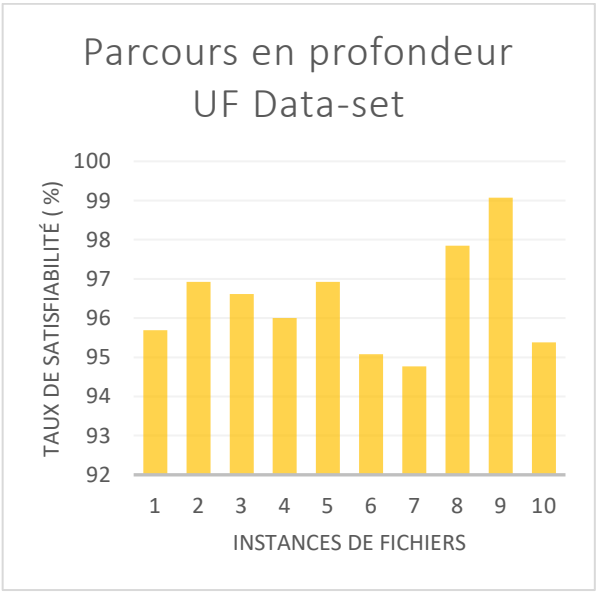


Figure3.2 taux satisfiabilité pour fichier uf75 et uuf75 parcours en profondeur

Pour ce parcours , les diagrammes presentés ci dessus nous apportent des taux assez important allant jusqu'a 94% . Ce qui prouve l'efficacité relative de cette methode.

Pour compléter notre étude expérimentale et bien comprendre le fonctionnement du parcours DFS , nous avons limité la profondeur à 25 , 50 sur 5 fichiers uf75

	Taux de satisfiabilité Moyen 25	Taux de satisfiabilité Moyen 50	Taux de satisfiabilité Moyen 75
uf75-01.cnf	55.23%	65.23	89.23
uf75-02.cnf	66.52%	66.76	94.06
uf75-03.cnf	52.60%	72.39	91.63
uf75-04.cnf	57.32%	81.69	93.23
uf75-05.cnf	49.21%	77.12	92.92

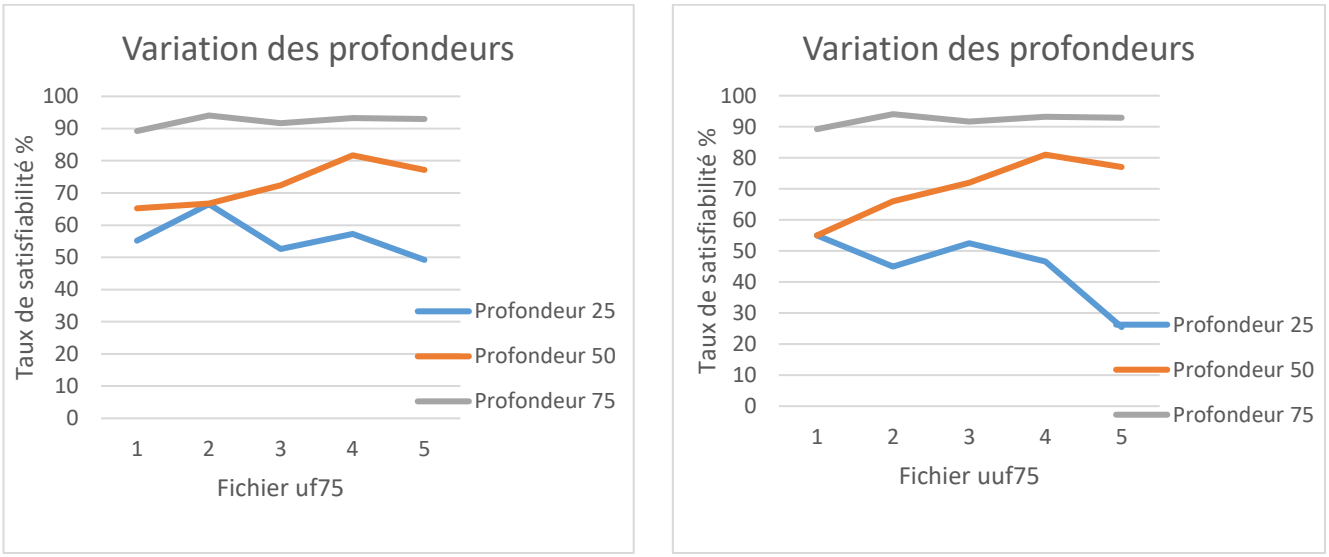


Figure3.3 taux satisfiabilité pour fichier uf75 et uuf75 parcours en profondeur d'abord avec variation de la profondeur

On déduit que le nombre de clauses satisfiables est proportionnel à la profondeur fixée.

3/ Cout uniforme :

Nom du fichier	Nombre de clauses satif Moyen **sur 10 exécutions**	Nombre de clauses satif minimal	Nombre de clauses satif maximal	Taux de satisfiabilité moyen
uf75-01.cnf	312	310	316	96.00%
uf75-02.cnf	319	315	321	98,15%
uf75-03.cnf	309	307	316	95,08%
uf75-04.cnf	312	306	315	96.00%
uf75-05.cnf	315	312	318	96,92%
uf75-06.cnf	316	314	317	97,23%
uf75-07.cnf	315	312	316	96,92%
uf75-08.cnf	315	311	321	96,92%
uf75-09.cnf	315	309	316	96,92%
uf75-010.cnf	310	304	312	95,38%

Nom du fichier	Nombre de de clauses satif Moyen **sur 10 executions**	Nombre de de clauses satif minimal	Nombre de de clauses satif maximal	Taux de satisfiabilité moyen
uuf75-01.cnf	311	309	313	95,69%
uuf75-02.cnf	315	312	318	96,92%
uuf75-03.cnf	314	311	315	96,62%
uuf75-04.cnf	312	308	313	96.00%
uuf75-05.cnf	315	306	318	96,92%
uuf75-06.cnf	309	303	312	95,08%
uuf75-07.cnf	308	307	312	94,77%
uuf75-08.cnf	318	311	320	97,85%
uuf75-09.cnf	322	321	322	99,08%
uuf75-010.cnf	310	309	315	95,38%

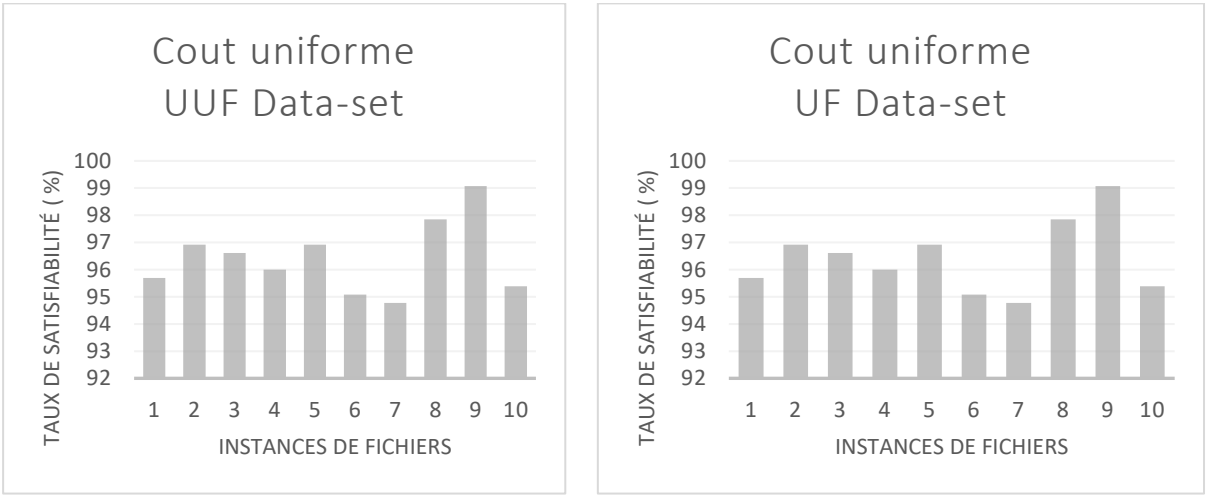


Figure3.4 taux satisfiabilité sur fichier uf75 et uuf75 en utilisant cout uniforme

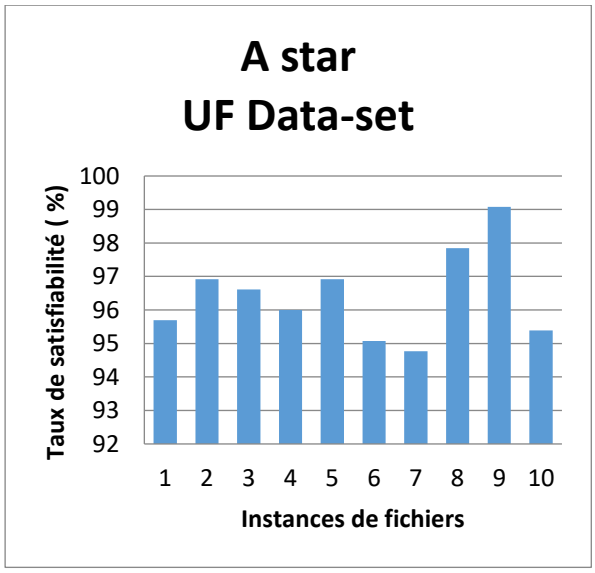
Pour ce parcours , les diagrammes présentés ci dessus nous apportent des taux satisfaisant allant jusqu'a 99,08% pour les fichier uuf75 soit 322 clauses satisfaites ainsi qu' un taux de 98,15% pour les fichiers uf75 . Ce qui prouve l'efficacité relative de cette méthode.

4/ A star :

Comme dit plus haut **l'heuristique** que nous avons utilisé est la fréquence d'apparition de chaque littéraux dans le fichier testé.

Nom du fichier	Nombre de clauses satif Moyen **sur 10 exécutions**	Nombre de clauses satif minimal	Nombre de clauses satif maximal	Taux de satisfiabilité moyen
uf75-01.cnf	312	306	321	96.00%
uf75-02.cnf	319	312	320	98,15%
uf75-03.cnf	309	301	315	95,08%
uf75-04.cnf	312	306	316	96.00%
uf75-05.cnf	315	309	318	96,92%
uf75-06.cnf	316	310	318	97,23%
uf75-07.cnf	315	311	318	96,92%
uf75-08.cnf	316	309	319	97,23%
uf75-09.cnf	315	312	321	96,92%
uf75-010.cnf	310	308	316	95,38%

Nom du fichier	Nombre de clauses satif Moyen **sur 10 exécutions**	Nombre de clauses satif minimal	Nombre de clauses satif maximal	Taux de satisfiabilité moyen
uuf75-01.cnf	313	309	316	96,31%
uuf75-02.cnf	315	311	32	96,92%
uuf75-03.cnf	314	313	315	96,62%
uuf75-04.cnf	312	310	316	96.00%
uuf75-05.cnf	316	312	317	97,23%
uuf75-06.cnf	309	307	311	95,08%
uuf75-07.cnf	313	310	317	96,31%
uuf75-08.cnf	318	311	321	97,85%
uuf75-09.cnf	322	316	325	99,08%
uuf75-010.cnf	310	309	316	95,38%



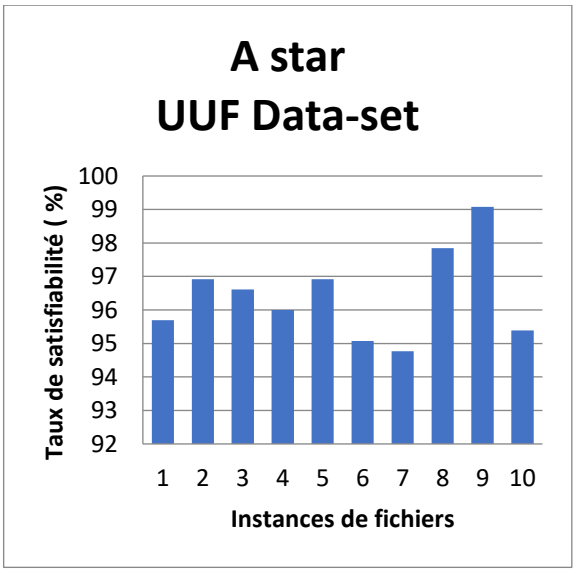


Figure3.5 taux satisfiabilité pour fichier uf75 et uuf75 parcours A*

Pour ce parcours , les diagrammes présentés ci dessus nous apportent des taux satisfaisant allant jusqu'a 99,08% pour les fichier uuf75 soit 322 clauses satisfaites ainsi qu un taux de 98,15% pour les fichiers uf75 . Ce qui prouve l'efficacité relative de cette méthode.

* Dans le but d'approfondir nos recherche nous avons voulu essayer une deuxième heuristique qui consistera cette fois à sommer la fonction avec la profondeur du chemin testé.

Nom du fichier	Nombre de de clauses satif Moyen **sur 10 executions**	Taux de satisfiabilité moyen
uf75-01.cnf	312	96.00%
uf75-02.cnf	319	98,15%
uf75-03.cnf	309	95,08%
uf75-04.cnf	312	96.00%
uf75-05.cnf	315	96,92%
uf75-06.cnf	316	97,23%
uf75-07.cnf	315	96,92%
uf75-08.cnf	315	96,92%
uf75-09.cnf	315	96,92%
uf75-010.cnf	312	96.00%

Nom du fichier	Nombre de de clauses satif Moyen **sur 10 executions**	Taux de satisfiabilité moyen
uuf75-01.cnf	311	95,69%
uuf75-02.cnf	315	96,92%
uuf75-03.cnf	314	96,61%
uuf75-04.cnf	312	96.00%
uuf75-05.cnf	315	96,92%
uuf75-06.cnf	309	95,08%
uuf75-07.cnf	308	94,76%
uuf75-08.cnf	318	97,85%
uuf75-09.cnf	322	99,08%
uuf75-010.cnf	310	95,38%

Nous constatons qu'avec cette seconde heuristique les résultats sont similaire a ceux obtenus grâce à la première heuristique

d. Comparaison entre les 4 méthodes:

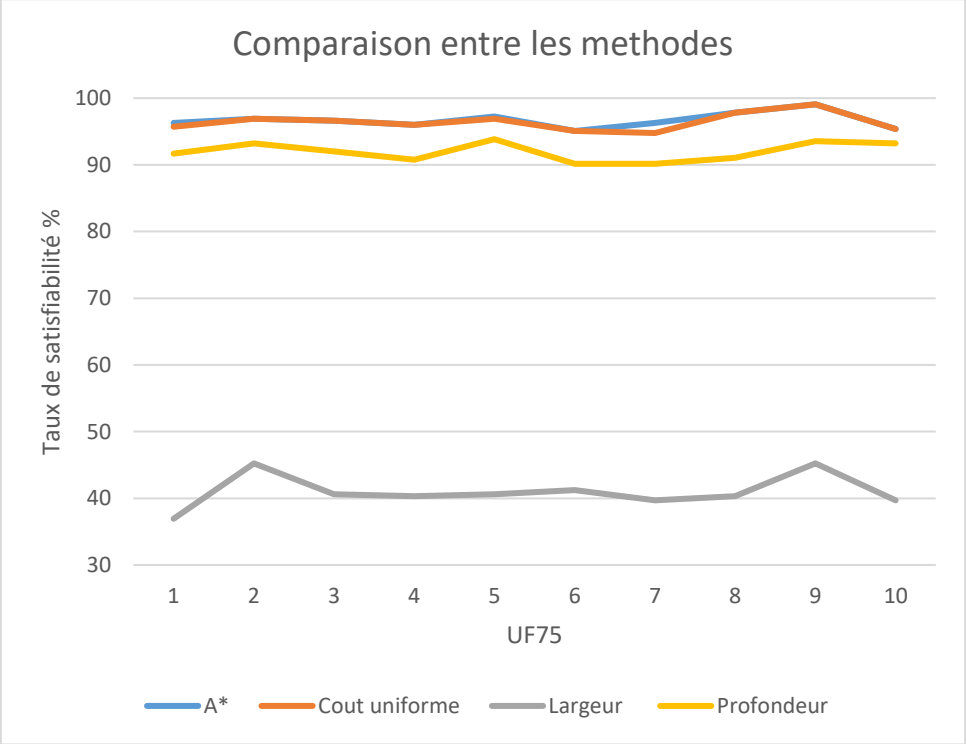


Figure3.6 comparaison du taux de satisfiabilité entre les 4 méthodes

On remarque clairement d'après les résultats du graphe l'importante différence entre les méthodes utilisées .

La moins performante est bien la méthode de recherche en largeur d’abord ,ayant les plus grandes complexité temporelle et spatiale . Elle atteint au maximum un taux de 45.53% .

La méthode de recherche en profondeur nous apporte de meilleurs résultats allant jusqu'à 94.06%. Mais toujours pas assez efficace en complexité temporelle.

Quant aux méthodes A* et cout uniforme qui apportent des résultats assez proches , on remarque une nette amélioration allant jusqu’ 99,08% soit 322 clauses satisfiables .La complexité temporelle et spatiale sont aussi assez réduites.

La similitude observée entre les deux dernières méthodes revient au fait que l'heuristique utilisée n'est pas déterminante.

Conclusion:

Cette premier parti du projet nous a permis de mettre l'accent sur le faible taux d'efficacité des méthodes de recherche utilisées, que ce soit les méthodes aveugles (DFS,BFS, cout uniforme), ou les méthodes informées (A*) en ce qui concerne les problème de taille importante, en l'occurrence dans notre cas, le SAT. Ces dernières sont bien trop couteuse en matière de mémoire et de temps.

Pour cela nous allons explorer des alternatives bien plus performante dans la partie suivante.

Partie II
Utilisation de BSO pour la résolution du problème de satisfiabilité

1.Introduction à l'intelligence en essaim:

À ses débuts, l'Intelligence Artificielle a puisé son inspiration dans le comportement individuel de l'être humain, en cherchant à reproduire son raisonnement. Elle s'est donc focalisée sur la manière de représenter les connaissances d'un expert et de modéliser son processus de décision, pour construire des systèmes dont le résultat pouvait être qualifié d'« intelligent ».

Puis les chercheurs se sont inspirés d'un autre modèle d'intelligence qui est l'intelligence en essaim . Cette dernière est inspirée du comportement des sociétés d'insectes telles que les fourmis , les abeilles , les chauve-souris … etc.

Elle consiste donc à étudier et à construire des sociétés d'individus artificiels simples qui sont capables collectivement de fournir une réponse complexe.

L'intelligence en essaim a des applications dans de nombreux domaines tels que l'optimisation combinatoire , l'optimisation des fonctions numériques , l'industrie du divertissement, le datamining, le routage dans les réseaux et le web.

Pour notre projet nous nous intéressons à l'intelligence en essaim inspirée des abeilles , plus précisément nous allons utiliser une métaheuristique « Bees Swarm Intelligence : BSO » et l'appliquer au « Problème de satisfiabilité : SAT ».

a.Notion de metaheuritisque :

Le mot métaheuristique est dérivé de la composition de deux mots grecs:

- Heuristique qui vient du verbe heuriskein (ευρισκειν) et qui signifie 'trouver'
- Meta qui est un suffixe signifiant 'au-delà', 'dans un niveau supérieur'.

Plusieurs définitions ont été proposées parmi lesquelles on peut citer :

"A metaheuristic is formally defined as an iterative generation process which guides a subordinate heuristic by combining intelligently different concepts for exploring and exploiting the search space, learning strategies are used to structure information in order to find efficiently near-optimal solutions."

[I.H. Osman and G. Laporte, Metaheuristics: a bibliography. Annals of Operations Research 63, 513-623, 1996]

En résumé les metaheuristiques servent à résoudre des problèmes dont la solution n'est pas trouvée algorithmiquement vu la complexité sémantique . Elles ne sont pas spécifiques à un type de problèmes mais plutôt générales et permettent d'obtenir une des meilleures solutions en un temps record .

Les métaheuristiques sont renommées pour leur efficacité dans la résolution des problèmes d'optimisation ainsi que SAT auquel nous nous intéressons.

b. Description du comportement des abeilles:

La société des abeilles adapte un système de communication assez complexe permettant une bonne communication et une bonne performance .

L'abeille qui trouve du nectar retourne à la ruche et en partage une partie avec les autres abeilles. Pour communiquer avec les autres abeilles elle effectue des mouvements qui forme une danse circulaire ou autre. Cette danse peut être dense , accélérée ou non selon plusieurs critères parmi lesquels on cite la quantité , la distance et la direction de la source.

A chaque abeille est attribuée une zone de recherche dans le voisinage de cette source . Chacune apporte de nouvelles informations de retour à la ruche et en informe les autres abeilles : Cela permet de trouver les meilleurs endroits pour la récolte et ainsi d'avoir une meilleure production .

c.Principe de BSO (Bee SwarmOptimization) :

Citée plus d'une centaine de fois dans la littérature informatique la metaheuristique BSO (Bee SwarmOptimization) a été développée en 2005 par H. Drias, S. Sadeg et S. Yahi au niveau de l'USTHB. Elle s'inspire du comportement d'abeilles dans la phase de recherche de la nourriture.

- Détermination de la première solution Sref de manière aléatoire et affectation à BeeInit.
- A partir de Serf et des paramètres empiriques , une zone de recherche SearchArea est créée . Chaque point de cette zone est affectée à une abeille
- Chaque abeille explore sa zone affectée ainsi que son voisinage et l'ajoute à la liste Tabou
- Chaque abeille détermine la meilleure zone est l'ajoute à la table Dance.
- Une fois que toutes les abeilles aient fini leur exploration , on affecte la meilleure zone à Sref pour une nouvelle exploration
- La procédure est réitérée jusqu'à trouver la solution optimale ou l'atteinte du nombre maximal d'itération. Cela sans ré explorer une même zone grâce a la liste Tabou.

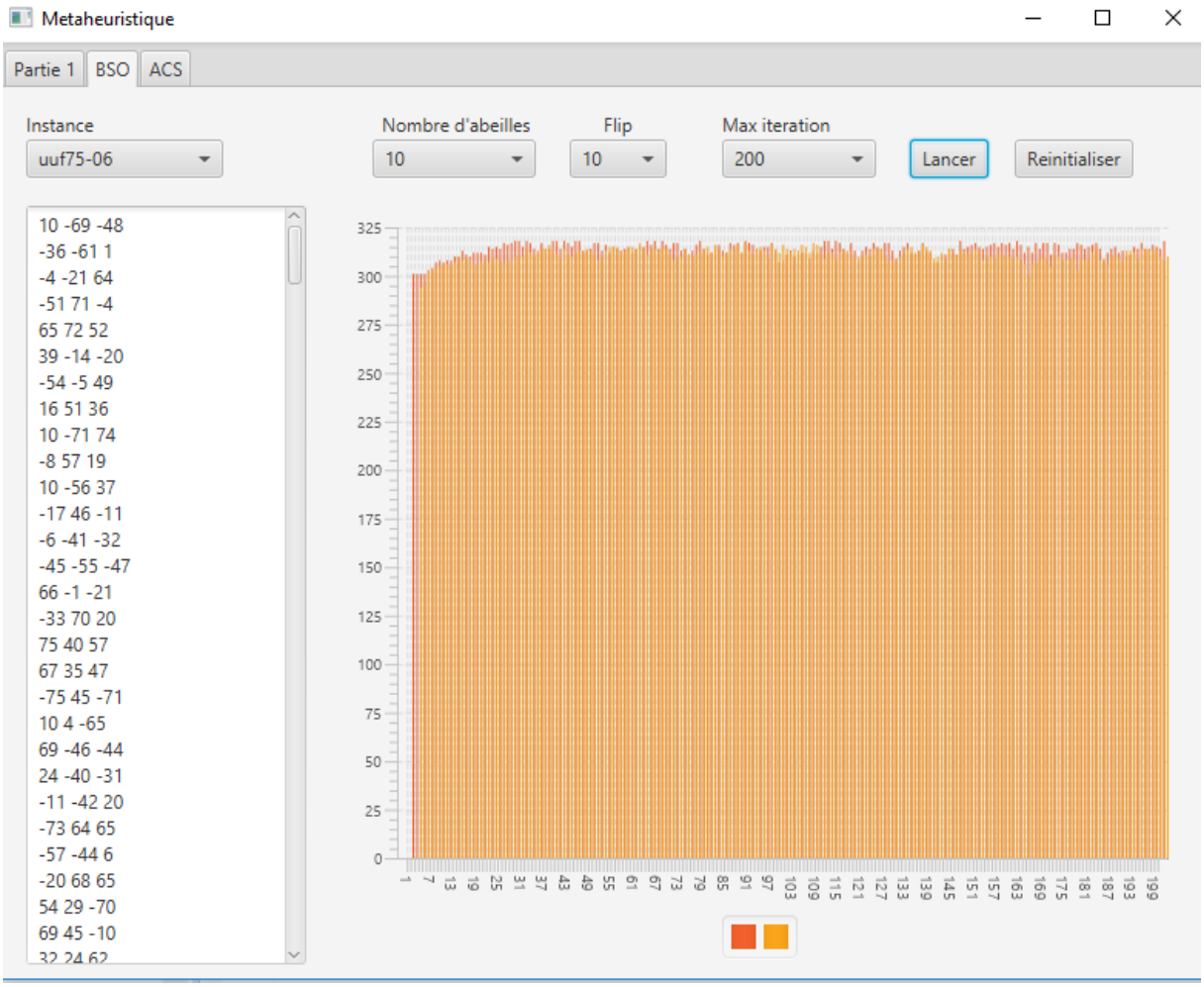
❖ **Algorithme BSO :**

```
Begin
Sref=Find Solution ( BeeInit) ;
While ( MaxIter not reached) and (not optimal)
{
        Insert (Taboo list , Sref ) ;
          Determine SearchPoints from Sref ;
        Assigna solution of SearchPoints for each bee ;
        For each (Bee i )
{ Search starting with the assigned solution ;
  Store ( Dance ) ;
}
Sref =  the best solution ;

}
End.
```

2. Résultats expérimentaux

A. Interface:



Notre interface affiche un histogramme représentant le taux de satisfiabilité de l instance choisie

avec les paramètres sélectionnés par l'utilisateur

b. déterminer les bons paramètres

Afin de trouver les meilleurs résultats nous avons varié les paramètres empiriques dans l'ordre suivant : Nombre d'itérations, Flip, Nombre d'abeilles , Max-chance.

Remarque: après avoir fixer la valeur d'un paramètre empirique nous utilisons cette valeurs dans la recherche de la valeur du prochain paramètre.

*Nous avons effectuer les test sur les fichier Benshmarks satisfiables:

A. <u>Variation du nombre d'itération :</u>

Nb itérations max	100	200	250	300	350	400
uf75-01	323	324	325	322	324	324
uf75-02	323	324	322	324	324	325
uf75-03	323	322	322	322	323	322
uf75-04	321	321	322	322	322	323
uf75-05	323	324	324	323	323	323
uf75-06	320	322	323	323	324	322
uf75-07	320	322	323	323	324	323
uf75-08	319	322	321	322	322	323
uf75-09	321	324	323	323	324	323
uf75-010	324	324	324	324	323	324
moy	321,7	322,9	322,9	322,8	323,3	323,2

Conclusion : On fixe le nombre **d'itération max** a **350.**

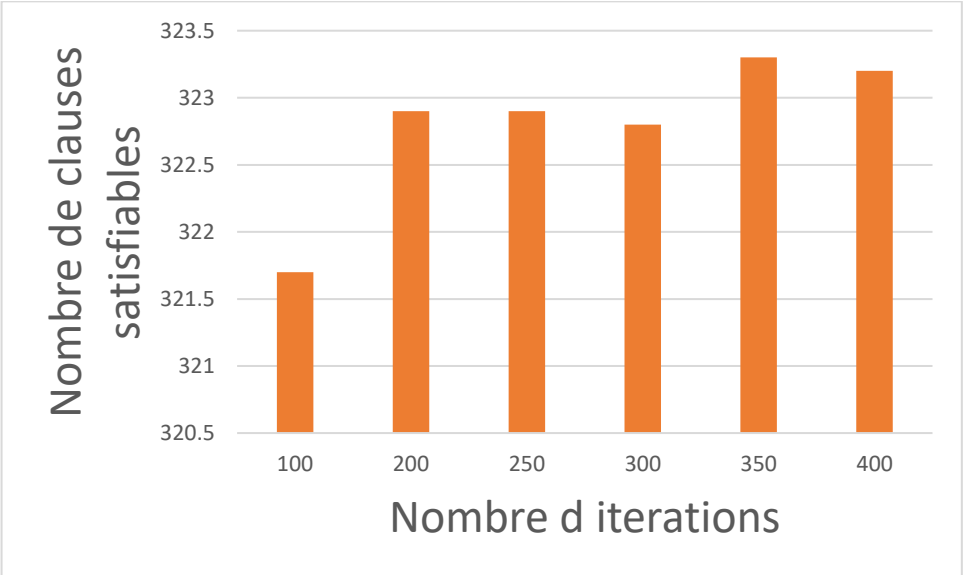


Figure 1Moyennes des taux de satisfiabilité en fonction du nombre d'itérations pour les instances UF75

B. Variation du flip :

Flip	5	10	15	20	22
uf75-01	315	320	324	325	322
uf75-02	317	323	324	323	323
uf75-03	318	321	321	323	322
uf75-04	317	320	323	322	323
uf75-05	316	324	323	324	323
uf75-06	317	318	318	323	324
uf75-07	315	322	321	322	322
uf75-08	315	318	323	323	323
uf75-09	319	321	324	323	323
uf75-010	316	322	324	324	324
moy	316,5	320,9	322,5	323,2	322,9

Conclusion : On fixe le flip a **20**.

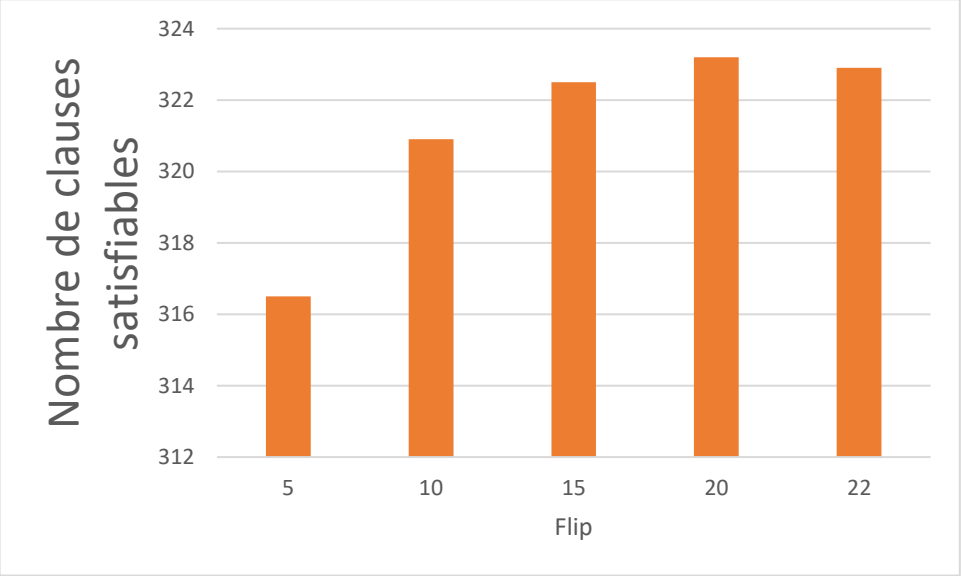


Figure 2 Moyennes des taux de satisfiabilité en fonction du flip pour les instances UF75

C. Variation du nombre d'abeilles :

Nb Abeilles	5	10	15	17	20
uf75-01	320	324	325	320	325
uf75-02	320	324	323	324	323
uf75-03	321	321	322	323	322
uf75-04	318	319	320	322	322
uf75-05	321	321	322	323	324
uf75-06	320	321	323	323	324
uf75-07	323	323	324	323	322
uf75-08	321	320	323	323	323
uf75-09	315	322	323	323	323
uf75-010	317	322	321	324	324
moy	319,6	321,7	322,6	322,8	323,2

Conclusion : On fixe le nombre d'abeilles à **20**.

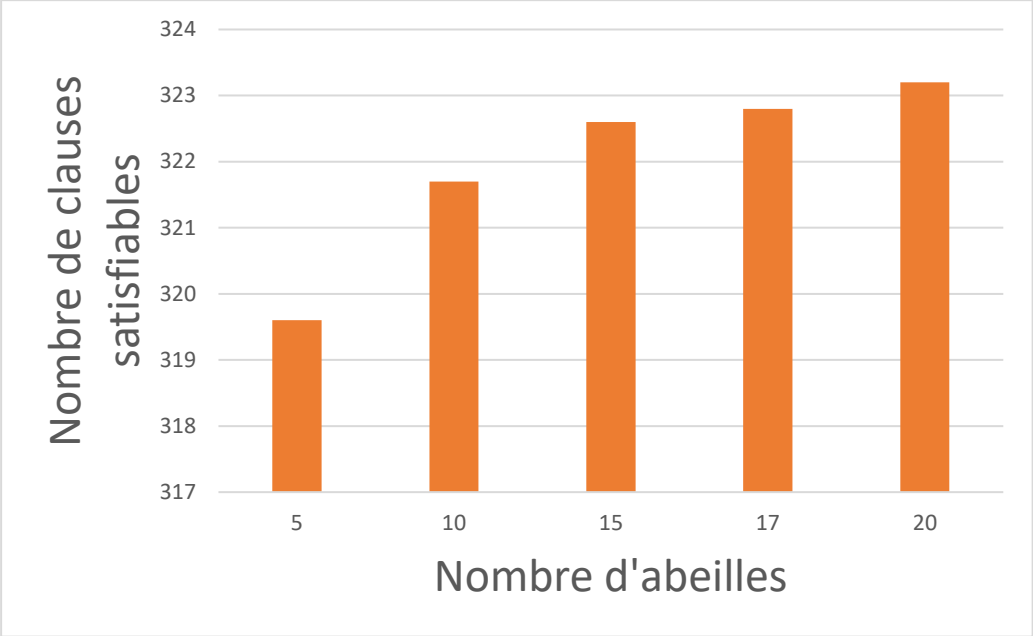


Figure 3Moyennes des taux de satisfiabilité en fonction du nombre d'abeilles pour les instances UF75

D. Variation du nombre de chances :

Nb Chance	5	10	15	20
uf75-01	325	325	322	323
uf75-02	324	325	325	324
uf75-03	324	324	321	322
uf75-04	320	323	323	322
uf75-05	323	325	323	322
uf75-06	322	324	323	320
uf75-07	323	325	320	323
uf75-08	323	324	322	322
uf75-09	322	325	323	323
uf75-010	324	324	324	324
moy	323	324,5	322,6	322,5

Conclusion : On fixe le nombre de chances a **10**.

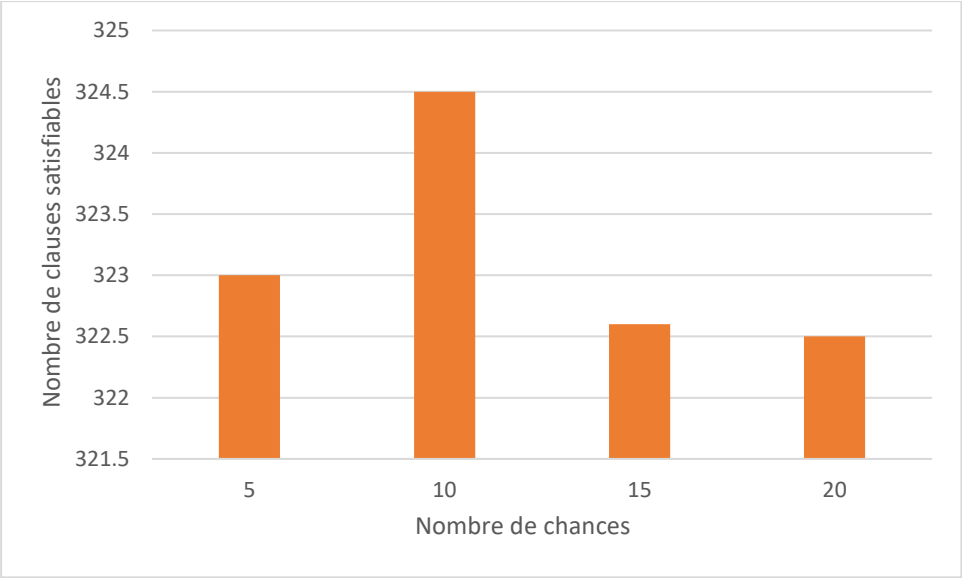


Figure 4 Moyennes des taux de satisfiabilité en fonction du nombre de chances pour les instances UF75

Une fois les paramètres empiriques fixés sont alors

Nombre itération max=350

flip=20

Nombre d'abeilles=20

Nombre de chance max=10

On remarque que le taux du nombre de clauses satisfiables

est très élevé et pour la majorité des instances le 100% est atteint en un temps très réduit .

Teste avec les bons paramètre sur les instances UF75:

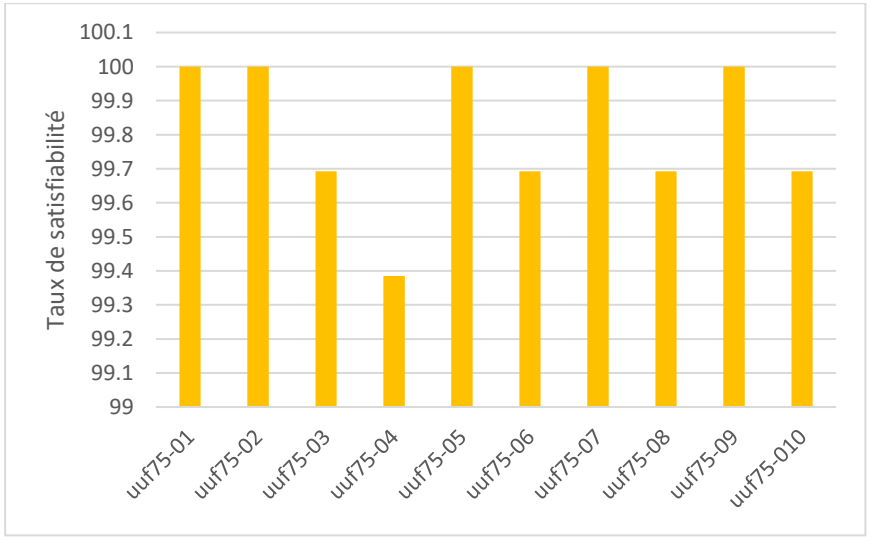


Figure 5 Taux de clauses satisfiables avec les meilleurs paramètres empiriques

Teste avec les bons paramètre sur les instances UUF75:

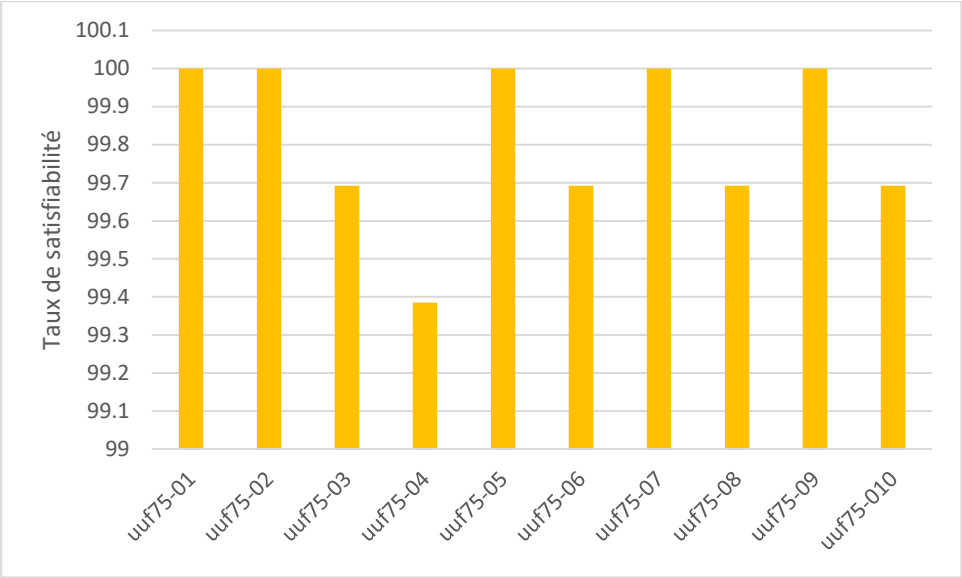


Figure 10 Taux de clauses non satisfiables avec les meilleurs paramètres empiriques

Conclusion :

La comparaison des résultats obtenus des tests effectués sur les algorithmes basés sur les méthodes aveugles et heuristiques, nous mène à conclure que les stratégies informées (utilisant une heuristique) sont les plus performantes et les plus optimales.

Par ailleurs, la recherche en largeur s'avère être la moins efficace pour le problème de satisfiabilité. En effet, elle se présente comme étant la plus coûteuse en espace mémoire.

Quant à la recherche en profondeur, on a constaté qu'elle est plus efficace que la recherche en largeur en termes d'espace mémoire. Mais, elle reste beaucoup moins optimale que la recherche utilisant une heuristique en termes de temps.

Ceci dit, ces résultats ne concernent que les espaces de recherches de taille moyenne ou de petite taille.

Pour le problème de satisfiabilité la taille de l'espace d'états est un facteur important. Quand les données de tests sont volumineuses, les méthodes aveugles ainsi que les méthodes heuristiques (qui sont des approches très performantes pour la résolution de problèmes complexes), deviennent inefficaces.

Partie III

Utilisation d'ACS pour la résolution du problème de satisfiabilité

1. Introduction

L'intelligence en essaim comme nous l'avons cité précédemment est basée sur le comportement collectif des systèmes auto-organisés. Les recherches sur les comportements collectifs des insectes sociaux fournissent aux informaticiens des méthodes puissantes pour la conception d'algorithmes d'optimisation. Les chercheurs informaticiens ont pu transformer des modèles du comportement collectif des insectes sociaux en méthodes utiles pour l'optimisation et le contrôle comme nous avons pu le voir précédemment avec BSO qui a donner d'excellents résultats. L'intelligence en essaim a pour objet de transformer la connaissance que les éthologistes en ont sur les capacités collectives de résolution de problèmes chez les insectes sociaux en techniques artificielles de résolution de problèmes. Parmi les algorithme développés en intelligence en essaim, nous verrons dans cette partie les algorithmes de contrôle et d'optimisation inspirés de modèles de recherche collective de nourriture chez les fourmis (ACO et ses dérivées ACS et AS), qui ont connu un certain succès et portent le nom d'optimisation par colonie de fourmis".

Dans ce travail de recherche nous allons présenter une vision de l'état de l'art des algorithmes de colonies de fourmis. Nous commençons d'abord par expliquer l'origine biologique de ces algorithmes. Nous présentons en suite le cadre ACO(optimisation par colonie de fourmis).

a. Moyen de communication des fourmis:

Les fourmis peuvent produire des produits chimiques parfumés également connus sous le nom de" phéromones". Les fourmis créent des phéromones à partir des glandes trouvées partout sur leurs corps. Leurs phéromones sont utilisées pour communiquer entre elles.

Les phéromones sont détectées à l'extrémité des antennes super sensibles des fourmis, les antennes gauches et droites indiquent à la fourmi le chemin à parcourir avec la force variable des phéromones. Les fourmis qui ont des antennes manquantes ou endommagées deviennent très désorientées.

Il y a environ dix à vingt parfums différents de phéromones (dépendant des espèces), chacun représentant un «mot chimique» que la colonie entière comprend. Les phéromones peuvent être utilisées pour invoquer quelques fourmis à des milliers de fourmis, selon ce qui est requis. Cela peut être l'attaque de proies, la défense de la colonie, la relocalisation de la colonie, ou l'emplacement d'une source de nourriture.

En effet, Au départ les fourmis ont plusieurs chemin menant a une source de nourriture, le choix se fait alors de manière complètement aléatoire. Lors de son retour au nid, la fourmi va de nouveau emprunter au hasard un chemin en déposant de la phéromone tout au long de son trajet. Et cette phéromone va ensuite biaiser le choix des autres fourmis qui circulent . Au cours du temps, de petites fluctuations de densité de phéromone vont apparaître sur le chemin le plus traversé , cette concentration élevée de phéromone indique alors le meilleur chemin reliant le nid à la source de nourriture.

b. Principe de ACO et ACS :

La première métaheuristique d'optimisation par essaim de fourmis a été initialement proposéeen1990 par Marco Dorigo et al. , afin de rechercher les chemins les plus optimaux dans un

graphe (voyageur du commerce). La métaheurtique a été plus tard utilisée pour la résolution de plusieurs problèmes d'optimisation, et plusieurs versions, s'inspirant des divers aspects du comportement des fourmis, dont ACS, ont vu le jour.

Lorsqu'une fourmi doit prendre de décision sur la direction à prendre, elle doit choisir le chemin ayant la plus forte concentration en phéromone, c'est-à-dire la décision dépend de la probabilité de transition d'un emplacement à une autre (une règle de transition est utilisée). Cette probabilité dépend de la concentration en phéromone. Chaque fourmi est considérée comme un agent capable de générer des solutions. Un mécanisme d'évaporation est utilisé afin d'oublier les anciens régions visités et se concentrer sur d'autres nouvelles. Des « update rules » sont utilisées pour cela.
Une fourmi meurt après avoir construit une solution et mis à jour la phéromone. A la fin de chaque itération de l'algorithme, un taux de phéromone est ajouté à la meilleure solution trouvée à ce niveau. Cette technique est appelée « offline update ».
ACS est l'une des variantes les plus connues de cette métaheuristique. Elle est basée sur l'introduction d'une nouvelle règle de déplacement appelée « règle pseudo-aléatoire proportionnelle » qui s'ajoute au processus de mise à jour locale des éléments des pistes de phéromones. Son objectif est de ce mécanisme est d'augmenter la diversification de la recherche.

Algorithme général :

Début
1- Initialisation des paramètres des regèles de transition et de mise à jour de la phéromone
2- Génération de k fourmis.
3-Fixation du nombre d'itération (condition d'arrêt de l'algorithme)
*4-**Tant qu'on** n'a pas atteint de nombre d'itérations fixé faire*
    ***Pour** chaque fourmi faire*
        ***Construire une solution** ;*
        Evaluer la solution ;
        Appliquer la mise à jour de phéromone « online delayed update » ;
    Fin
    Choisir la meilleure solution trouvée dans cette itération ;
    Appliquer la mise à jour de phéromone « offline delayed update » ;
Fin
Fin de l'algorithme.

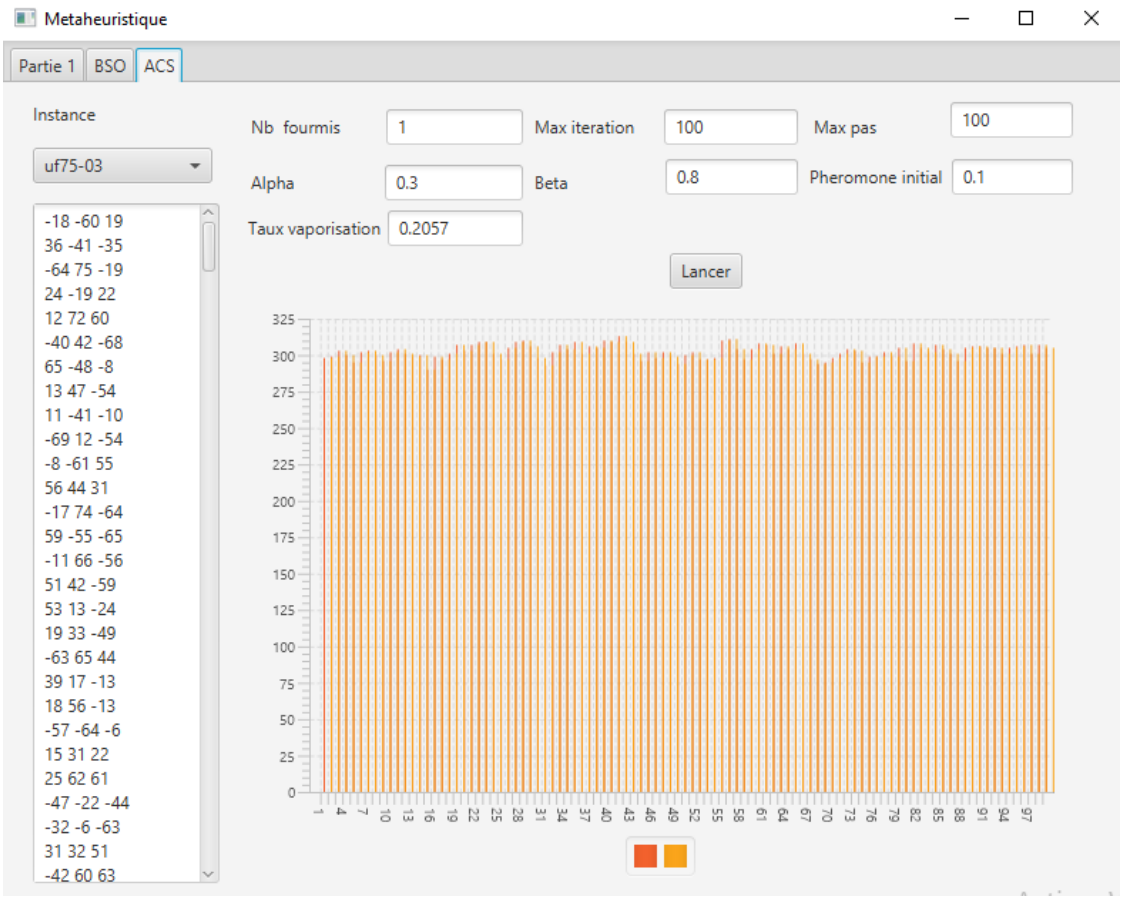
Fonction Construire une solution:

Entrée *: Liste des littéraux L*
Sortie *: solution S*
1- La solution S est initialement vide
2- ***Tant que*** *la liste des littéraux L n'est pas vide faire*
- *Utiliser la règle de transition pour choisir un littéral*
- *Supprimer le littéral et son contraire de L*
- *Ajouter le littéral sélectionné à la solution S*

Fin
Fin de la construction *(retourner S) ;*

2.Etude expérimentale et résultats :

a. Interface



Nous pouvons voir sur notre interface un histogramme représentant le taux de satisfiabilité pour l instance choisie avec les paramètres sélectionnées par l'utilisateur

b. fixer les valeurs des paramètres

❖ Variation du nombre de fourmis

Nombre fourmis	20	30	40	50	60	70
uf75-01	296	311	300	300	302	305
uf75-02	301	305	296	299	300	301
uf75-03	300	306	301	301	300	300
uf75-04	300	301	299	296	300	310
uf75-05	306	300	298	301	298	300
uf75-06	308	309	300	310	301	311
uf75-07	302	305	307	308	306	307
uf75-08	308	306	310	308	310	309
uf75-09	308	302	307	309	307	306
uf75-010	305	311	308	312	312	311
moy	303,4	304,6	302,6	304,4	303,6	309,6

meilleur nombre fourmis = 70

❖ Variation de α

Nombre fourmis	0.4	0.5	0.7	0.9
uf75-01	300	300	300	315
uf75-02	299	309	298	313
uf75-03	301	305	301	312
uf75-04	296	306	306	296
uf75-05	300	302	300	311
uf75-06	308	296	300	312
uf75-07	302	301	307	308
uf75-08	309	300	309	309
uf75-09	308	300	306	309
uf75-010	305	296	308	312
moy	302,8	301,5	303,5	309,7

meilleur valeur de alpha est 0.9

❖ Variation de β :

Nombre fourmis	0.4	0.5	0.7	0.9
uf75-01	300	300	315	302
uf75-02	302	309	313	313
uf75-03	301	305	312	312
uf75-04	296	306	301	301
uf75-05	300	302	310	311
uf75-06	308	296	312	312
uf75-07	299	300	308	308
uf75-08	301	308	309	309
uf75-09	300	300	309	309
uf75-010	300	296	312	312
moy	300,7	302,2	310,1	308,9

meilleur résultat avec beta= 0.7

❖ variation de q0:

Nombre fourmis	0.2	0.5	0.8	0.9
uf75-01	301	311	296	312
uf75-02	311	312	300	309
uf75-03	312	308	308	310
uf75-04	308	309	299	312
uf75-05	301	311	310	311
uf75-06	308	296	312	312
uf75-07	300	300	308	312
uf75-08	308	308	309	311
uf75-09	300	303	309	309
uf75-010	299	296	312	312
moy	304,8	305,4	306,3	311,6

meilleur résultat avec q0= 0.9

❖ Variation de r0:

Nombre fourmis	0.1	0.2	0.3	0.4
uf75-01	311	308	314	312
uf75-02	309	309	313	309
uf75-03	310	311	315	310
uf75-04	296	296	312	309
uf75-05	300	311	315	312
uf75-06	308	296	315	311
uf75-07	310	300	312	309
uf75-08	312	308	311	312
uf75-09	308	303	313	309
uf75-010	300	308	314	311
moy	306,4	305	313,9	310,4

Meilleur résultat avec r0= 0.3

Le meilleur taux de satisfiabilité atteint est égal à 96,9% avec les paramètres suivants :
Nb_Fourmis = 70 ;
Nb_Itérations =250 ;
α = 0.9
β = 0.7
q0=0.9
r0=0.3

Tests sur les instances satisfiables :

Nom du fichier	Max-SAT	Taux de satisfiabilité
uf75-01.cnf	315	96,9%
uf75-02.cnf	315	96,9%
uf75-03.cnf	311	95,6%
uf75-04.cnf	312	96%
uf75-05.cnf	315	96,9%
uf75-06.cnf	311	95,9%
uf75-07.cnf	315	96,9%
uf75-08.cnf	314	96,2%
uf75-09.cnf	315	96,9%
uf75-010.cnf	312	96%

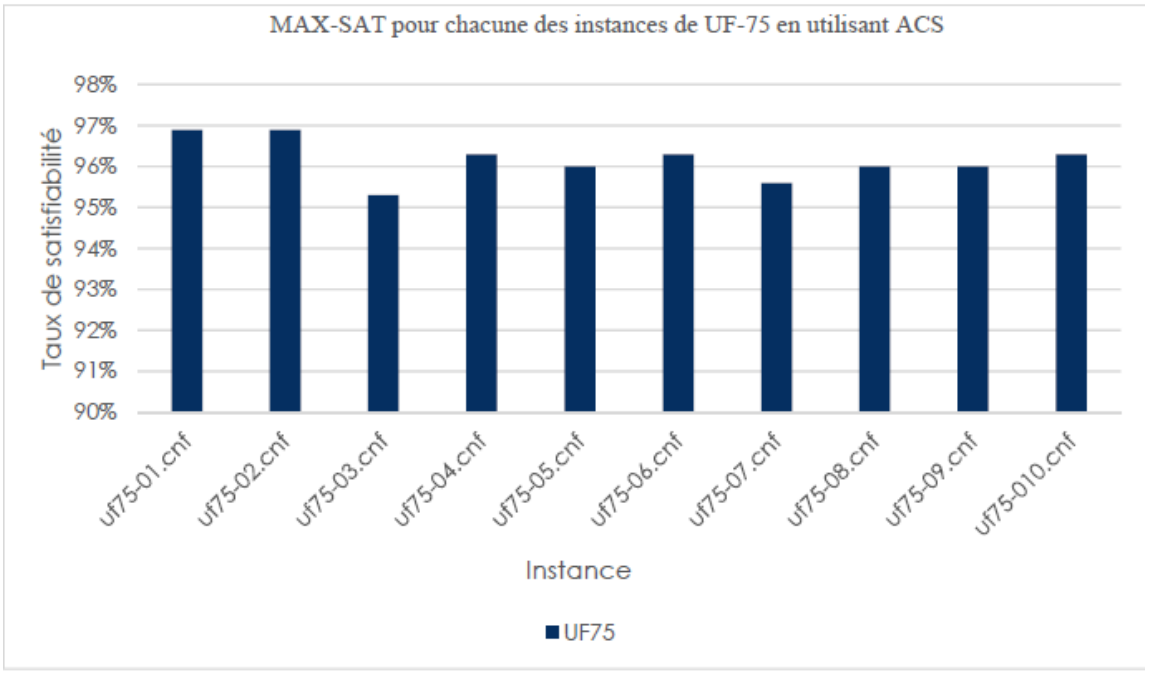


Figure 1: Histogramme montrant les résultats MAX-SAT de l'application de l'algoritme basé sur ACS sur chaque instance du fichier benchmark satisfiable UF-75

Tests sur les instances non satisfiables :

Nom du fichier	Max-SAT	Taux de satisfiabilité
uuf75-01.cnf	311	95,6%
uuf75-02.cnf	311	95,6%
uuf75-03.cnf	308	94,7%
uuf75-04.cnf	306	94.1%
uuf75-05.cnf	308	94.7%
uuf75-06.cnf	309	95%
uuf75-07.cnf	309	94,4%
uuf75-08.cnf	308	95,3%
uuf75-09.cnf	310	95,3%
uuf75-010.cnf	306	94.1%

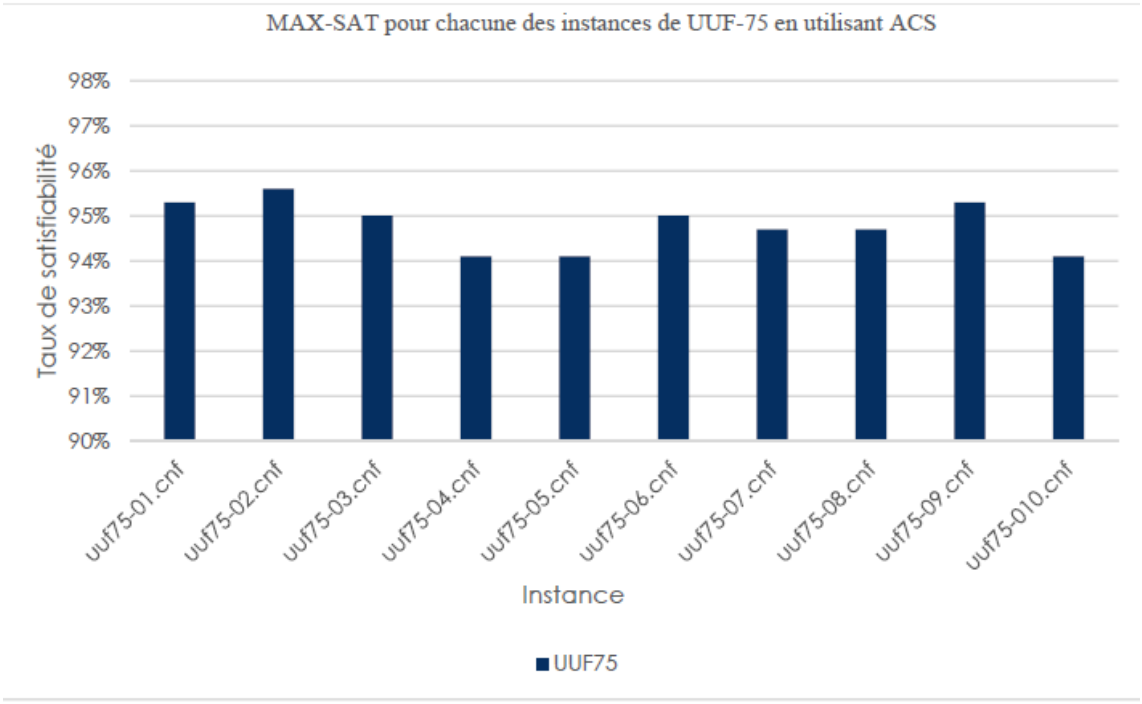


Figure 3 : Histogramme montrant les résultats MAX-SAT de l'application de l'algoritme basé sur ACS sur chaque instance du fichier benchmark insatisfiable UUF-75

3. Comparaison des résultats obtenus avec les méthodes de la première et deuxième partie :

a. Fichiers satisfiables :

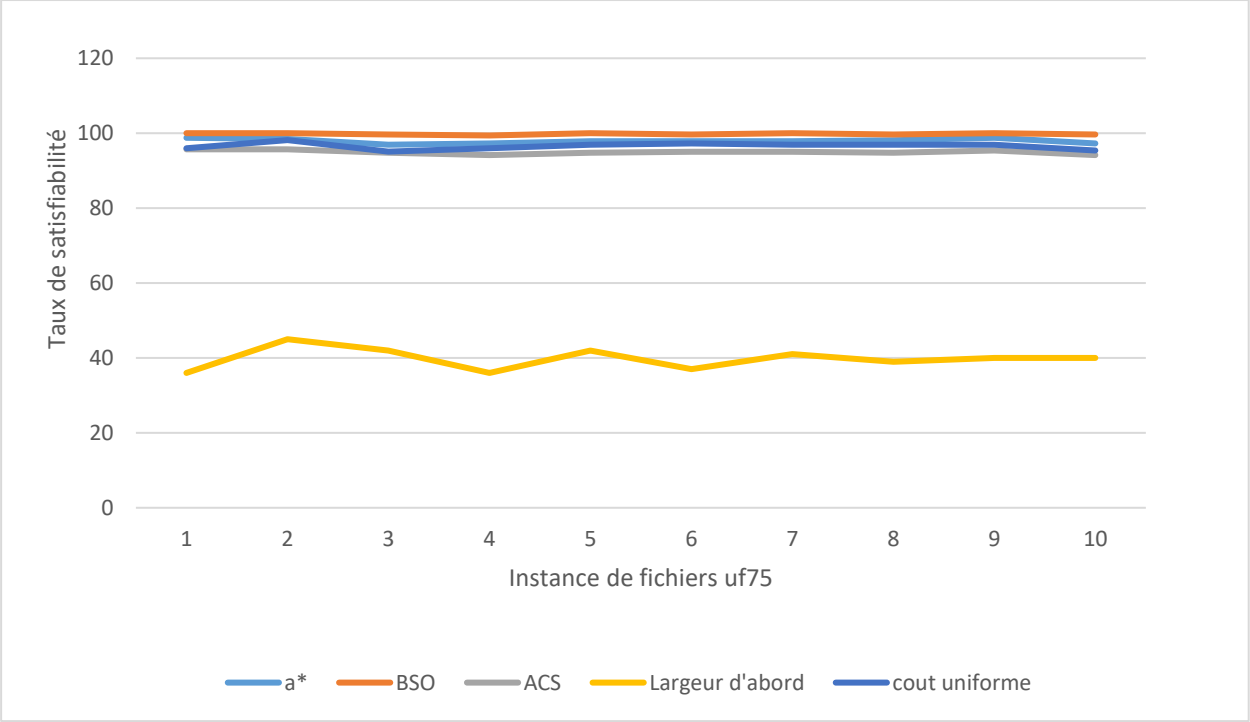


Figure 6: Comparaison entre les differentes méthodes pour les instances des fichiers UF75

b. Fichiers non satisfiables :

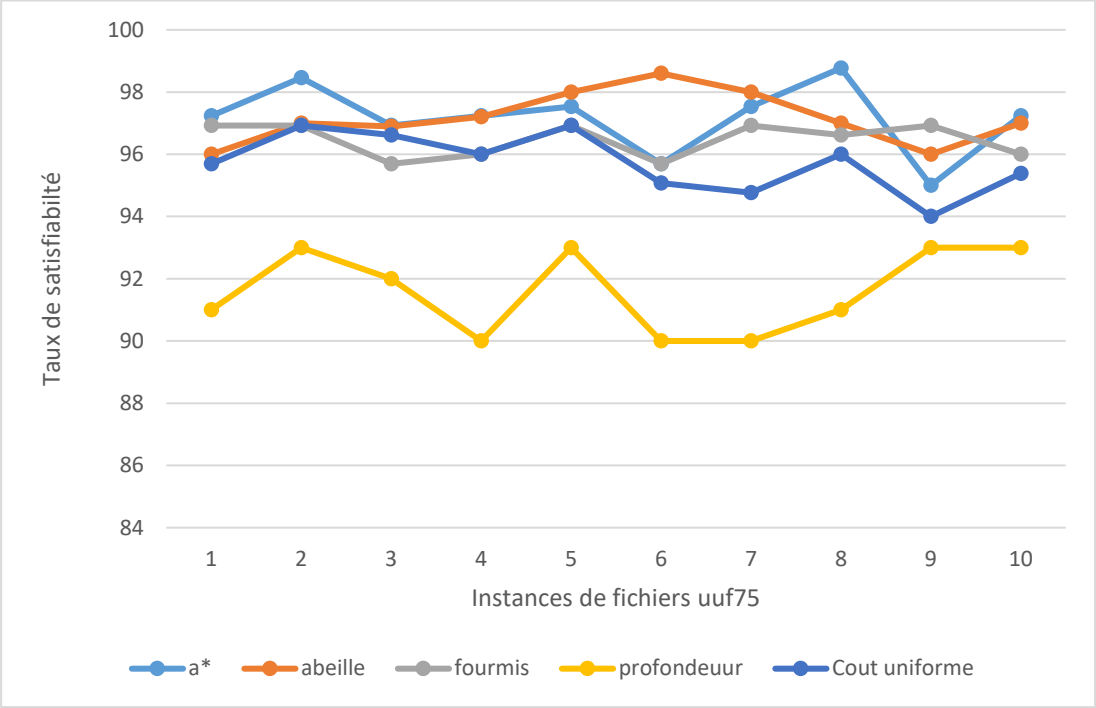


Figure 7: Comparaison entre les differentes méthodes pour les instances des fichiers UF75

Après observation et analyse des résultats obtenus des tests des trois algorithmes de résolutions sur les benchmarks choisis, on constate clairement qu'aucune des stratégies de recherche aveugle n'a pu atteindre le but dans l'intervalle de temps fixé .
Le recherche en largeur d'abord amène tout le temps à l'explosion combinatoire, ceci confirme le fait que cette stratégie est très gourmande en espace mémoire et est donc inefficace pour des données de grande taille.
La recherche en profondeur quant à elle, ne mène pas à une explosion de l'espace mémoire mais reste complexe en temps de recherche. En effet, pour un espace d'états de grande taille, il est très difficile d'atteindre un but.

L'algorithme A*, qui est à la base plus efficace que les deux autres stratégies, s'avère aussi ne pas être très efficace sur les grands espaces de recherche.
Même en essayant plusieurs heuristiques nous n'avons pas pu avoir le 100%. Ceci revient au choix de l'heuristique qui est dépendant du problème . C'est la ou réside la difficulté dans l'algorithme A* : l'heuristique est spécifique au problème .

On conclut que pour ce problème de satisfiabilité l'inefficacité des algorithmes de recherches aveugles et des simples heuristiques.

Pour les métaheuristiques (BCO et ACS) nous avons clairement eu de meilleurs résultats que les algorithmes de recherche précédant .
Nous avons pu atteindre le 100% pour plusieurs instances en utilisant BSO et un assez bon taux en .
Cela revient au fait que ces metaheuristiques soient flexibles pour tout type de problème, il suffit juste de modéliser le problème .

Autre que la performance , le facteur de temps est nettement meilleur en utilisant les métaheuristiques qui donnent des résultats en quelques secondes contrairement aux autres qui dépassent des dizaines de minutes (voire des heures pour le parcours en largeur d'abord) .

En comparant les résultats obtenus en utilisant les métaheuristiques BSO et ACS et avec les paramètres empiriques fixés , on remarque que BSO surpasse ACS et cela pour la plupart des instances de fichiers .

Conclusion générale

Cette partie du projet nous a permis de confirmer qu' il est souvent impossible de résoudre des problèmes de taille importante comme le problème SAT, avec des méthodes de résolution exhaustives ou purement algorithmiques, telles que les méthodes aveugles ou celles utilisant des heuristiques, car ces dernières sont coûteuses en termes de temps et/ou espace mémoire. Il est clair que l'utilisation d'autres méthodes, plus performantes comme les métaheuristiques est nécessaire dans ce cas.
Les résultats de ces approches prouvent leur indispensabilité, et leur bon augure pour progresser de manière significative dans la résolution des problèmes de grande taille.