République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Université des Sciences et de la Technologie Houari Boumediene

Faculté d'Electronique et d'Informatique
Département d'Informatique

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

Rapport du projet :

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

Réalisé par :

- **MEZABIAT Aimen Said**
- **BENCHABEKH Walid**
- **SEDKAOUI Amine**
- **FERFERA Oualid Khaled**

M1 - SII

I. Introduction générale :

Le problème SAT fut le premier problème à avoir été démontré comme un problème NP-Complet (par "Stephen Cook" en 1971). C'est donc le pionnier et l'ancêtre de tous les problèmes NP-Complets car à partir de SAT, la classe des problèmes NP-Complets a pu être concrétisée. Il consiste à décider si une formule de la logique propositionnelle, présentée en forme normale conjonctive (CNF), est satisfiable ou non. Ce problème est NP-complet. Par conséquent, tous les algorithmes connus sont de complexité exponentielle.
En particulier, les algorithmes à recherche aveugle qui consistent à énumérer toutes les évaluations des variables, sont couteuses en facteurs de consommation d'espace mémoire et de temps de calcul. De ce fait il fallait changer, améliorer éventuellement à l'aide des différentes heuristiques et encore des métas heuristiques.

Notre projet est divisé en deux parties, dans cette première partie, nous nous intéressons à deux stratégies : La recherche aveugle et la recherche informée, utilisant des heuristiques. Il s'agit d'implémenter des algorithmes de résolution du problème de satisfiabilité basés sur les deux méthodes citées précédemment, puis les tester sur des fichiers de benchmarks satisfiables et d'autres non-satisfiables, afin de comparer leurs complexité et efficacité.

Ensuite dans une deuxième partie du projet on va utiliser des algorithmes métas heuristiques voire « GA » (Genetic Algorithm) et « PSO » (Particle Swarm Optimization) en essayant toujours de trouver quel est le meilleur pour réduire la complexité de ce problème, et des autres problèmes similaires.

II. Description de l'espace de recherche

Consiste à attribuer pour chaque variable 1 ou 0 (c.-à-d. vrai ou faux), cela peut être modélisé par un arbre binaire dont les nœuds représentent l'assignation attribuer à une variable. Comme montrer dans la figure ci-dessous :

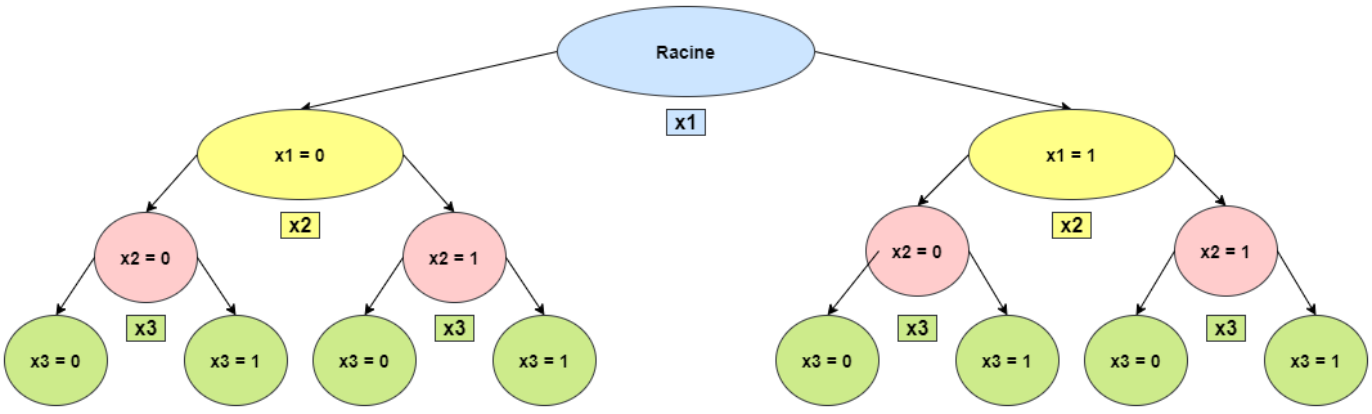


Figure 01 – Espace de recherche pour un ensemble de 3 variables

1) Méthodes aveugles

Les méthodes aveugles sont des méthodes qui consistent à énumérer tous les cas possibles jusqu'à trouver la bonne solution, Ainsi, toutes les combinaisons sont examinées aveuglement,

sans aucune information aidante à la prise de décision. Ces approches sont les plus basiques pour résoudre les problèmes dans l'intelligence artificielle. Parmi ces méthodes aveugles on retrouve la recherche en largeur et la recherche en profondeur.

1.1 Recherche en largeur d'abord

La recherche en largeur d'abord (BFS) consiste à examiner entièrement chaque niveau avant d'aller plus profondément dans la recherche. Par conséquence, les itérations de ce processus ne s'arrêtent que si un nœud apporte la solution (le cas de succès) ou que toutes les possibilités ont été examinées (le cas échéant). Cette stratégie garantie de trouver une solution si elle existe, mais elle est très gourmande en espace mémoire.

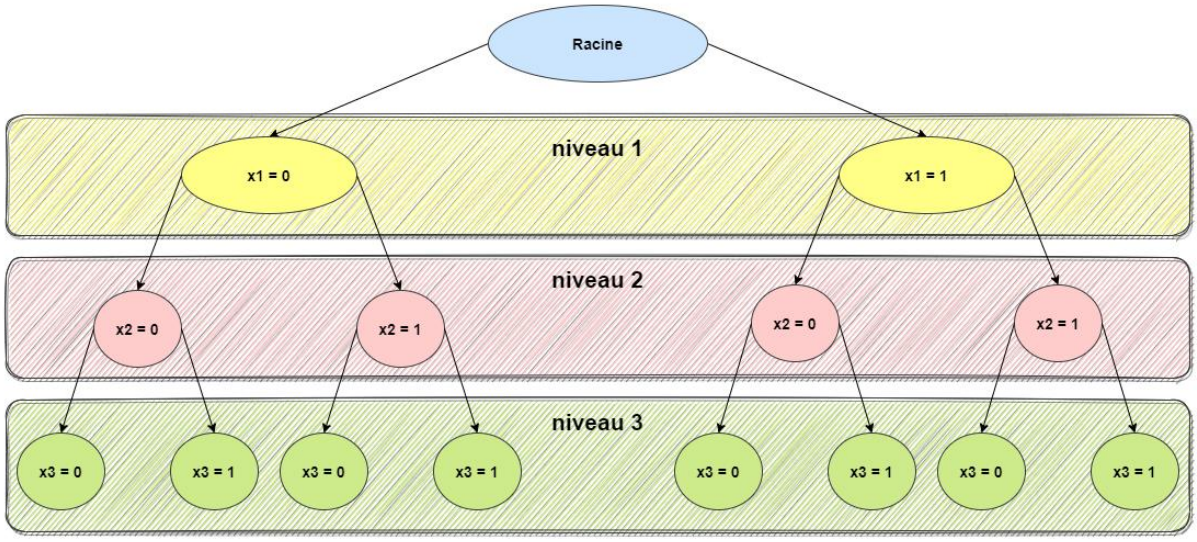


Figure 02 – Exemple du recherche BFS (ensemble de 3 variables)

1.2 Recherche en profondeur d'abord

L'idée de la recherche en profondeur d'abord (DFS) est d'explorer les branches de l'arbre une par une en vérifiant la satisfaction après chaque visite d'un nouvel nœud.
A noter que le choix des variables se fera aléatoirement et que le teste de la satisfaction sera vérifié à chaque visite d'un nouveau nœud (chaque assignation à la variable).
Cette stratégie est plus optimale en espace mémoire car elle s'occupe d'une seule branche avant d'aller aux autres i.e. elle ne mémorise pas tous les nœuds en mémoire, mais uniquement ceux du chemin menant au but.

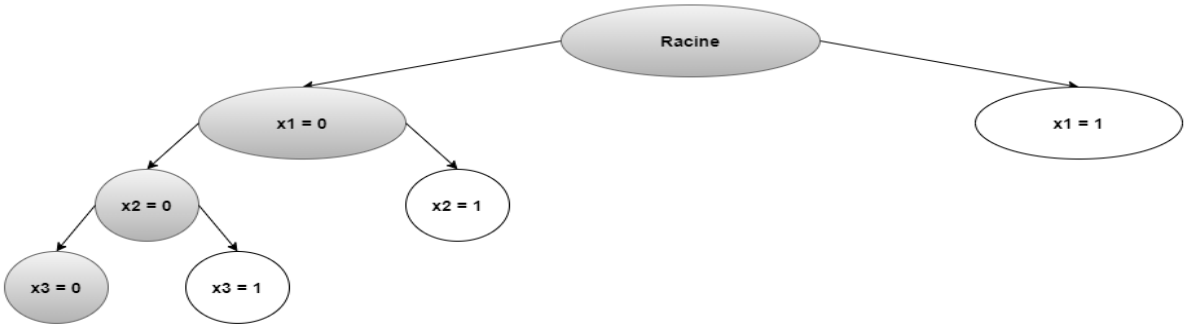


Figure 03 – Exemple du recherche DFS en une branche (ensemble de 3 variables)

2) Méthodes heuristiques

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

Algorithme A*

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

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

III. Le problème SAT

1) Description

Le problème SAT est un problème de logique propositionnelle, il consiste à décider si une **formule** en forme normale conjonctive est satisfiable, c'est-à-dire s'il existe une assignation de valeur de vérité (true, false) aux **variables** telle que toutes les **clauses** sont « true ».
Une variable propositionnelle est une variable booléenne qui peut être soit vraie ou fausse, elle est l'élément fondamental des **formules propositionnelles**. Cette dernière est une expression construite à partir de connecteurs logique (et, ou et non) et de variables propositionnelles.
On appelle une clause une disjonction de littéraux (forme normale disjonctive) tel qu'un littéral est une variable propositionnelle (littéral positif) ou sa négation (littéral négatif). Ainsi, le problème SAT est une conjonction d'un ensemble fini de clauses.

On définit une instance sur un nombre m de clauses Ci formées à l'aide de n variables propositionnelles (littéraux).
On note k-SAT la variante du problème SAT où toutes les clauses ont exactement k littéraux ($k >= 2$).
On s'intéresse dans la partie expérimentale de notre projet à la variante obtenue pour $k = 3$, et donc on s'intéressera aux instances 3-SAT qui sont un ensemble fini de clauses de même taille égale à trois littéraux, formées à l'aide de n variables propositionnelles.

2) Structure de données

Avoir de bonnes structures informatiques est nécessaire pour les performances de notre application. Pour cette raison nous avons choisi nos structures représentatives des entités du problème SAT voire la représentation d'une clause, ensemble de clauses, solution probable du problème SAT de sort à faciliter le déroulement du programme d'application.

a) Représentation d'une clause

La modélisation d'une clause a été implémentée sous forme d'un vecteur dynamique contenant "k" cases. k représente la variation du problème SAT et est connu au moment de l'exécution. Chaque case est un entier représentant l'indice du littéral, soit positif (i.e. valeur de vérité « *vrai* ») ou négatif (i.e. valeur de vérité « *faux* »).

Par exemple, pour un k = 3, soit la clause c_1 définit par $c_1 = \bar{x}_8 + \bar{x}_{15} + x_{25}$.
Cette clause est représentée par le vecteur

-8	-15	25

suivant :

b) Représentation d'un ensemble de clauses

Un ensemble de clauses est implémenté dans une matrice dynamique où chaque ligne représente une clause avec même raisonnement qu'une représentation d'une clause.

Exemple :

$c_1 = \bar{x}_8 + \bar{x}_{15} + x_{25}$

$c_2 = \bar{x}_{55} + x_{70} + x_{71}$ ⟶

$c_3 = x_1 + x_2 + \bar{x}_3$

-8	-15	25
-55	70	71
1	2	-3

c) Représentation d'une solution

On représente une solution potentielle d'un problème SAT avec un vecteur contenant autant de cases que le nombre de variables du l'instance donnée. Car une solution est un ensemble de valeurs associées aux ces variables permettant à satisfaire toutes les clauses de l'instance. Cette représentation facilite l'accès à un littéral de la solution à partir d'un littéral d'une clause car il suffit juste de prendre la valeur absolue de ce dernier et de retrancher « 1 » si les cases du vecteur solution commencent par l'indice « 0 », ce qui est le cas en général.

Si notre instance a 10 variables par exemple. Une solution potentielle est comme suit :

1	**-2**	**3**	**4**	**5**	**-6**	**-7**	**-8**	**9**	**10**

d) Fichier CNF

Afin d'obtenir l'ensemble des clauses et remplir notre structure de donnée citée plus haut (matrice) il faut d'abord récupérer les clauses depuis un **fichier cnf.**

Un **fichier cnf** est utilisé pour définir une expression booléenne écrite en forme normale conjonctive, il représente une instance du problème SAT, il se présente, en format ASCII, comme suit :

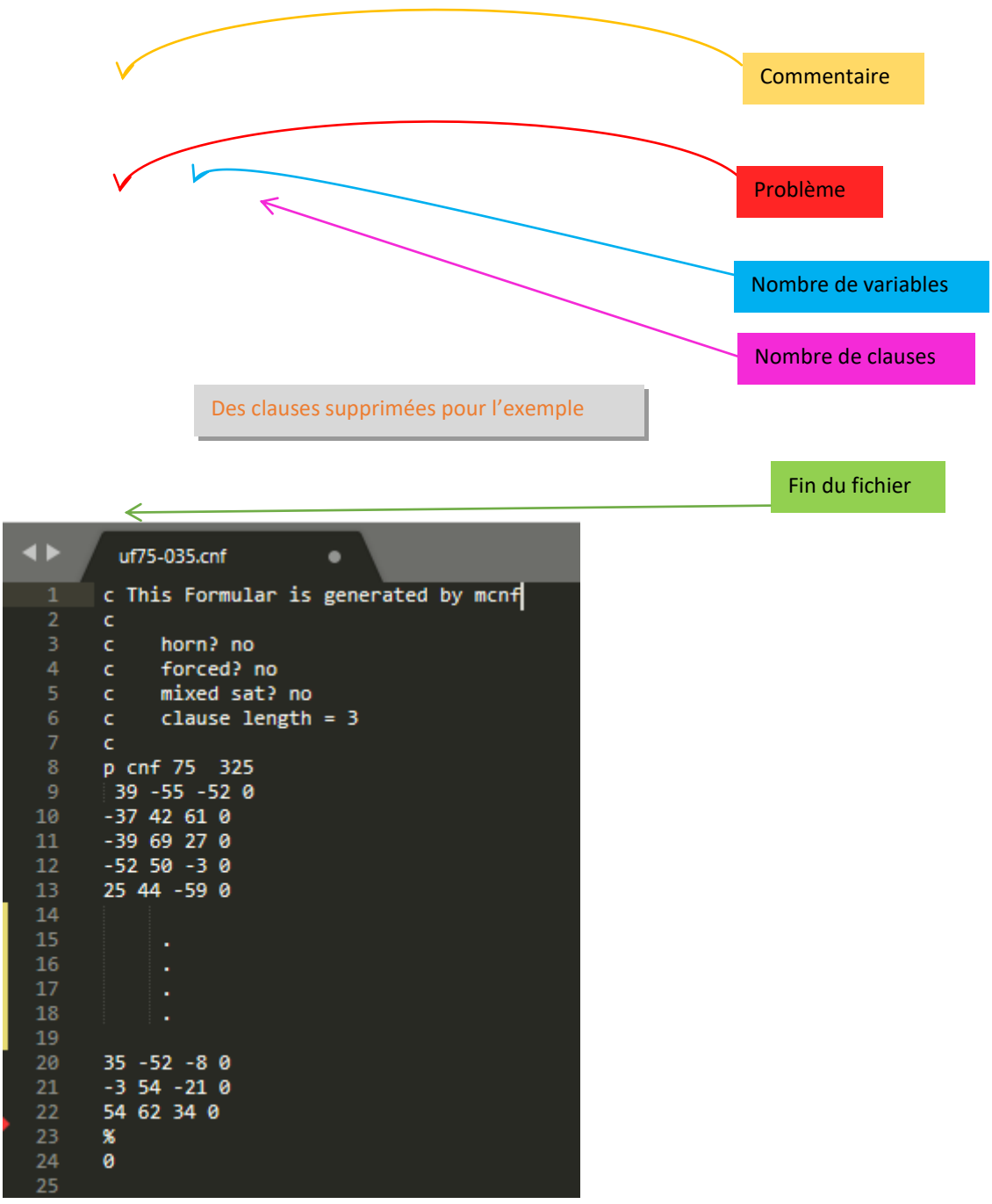


Figure 04 - Illustration explicatif d'un fichier CNF

Les fichiers cnf commence par des commentaires indiquant les caractéristiques de l'instance représentée. Cette ligne est identifiée par la lettre "c" minuscule. Puis la ligne du problème qui commence par la lettre "p" minuscule. Cette ligne contient les informations sur le nombre de variables et le nombre de clauses dans l'instance. Le reste des lignes représentent les clauses dont chacune se termine par un zéro sauf la dernière ligne qui contient le symbole "%" marquant la fin du fichier.

3) Fonction d'évaluation de la satisfiabilité d'une instance par une solution :

Evaluer la satisfiabilité d'une instance par une solution donnée S revient à déterminer le nombre des clauses satisfaites par cette solution. Pour cela on parcourt les clauses l'une après l'autre i.e. on parcourt la matrice de l'ensemble des clauses ligne par ligne, et pour chaque clause on vérifie s'il existe un littéral qui a son même correspondant dans la solution. Si cette correspondance existe dans la clause parcourut on dit que cette clause est satisfaite par la solution donnée. Cependant si la correspondance n'existe pas, on dit que la clause n'est pas satisfaite. On peut décrire cette fonction d'évaluation par l'algorithme suivant :

Algorithme Satisfiabilité;

Entrée :

M : La matrice contenant les clauses du problème SAT,

S : Solution correspondante aux paramètres du SAT;

Sortie :

Le nombre de clauses satisfaites par la solution ***S***;

```
Var : k_sat, nCl, nClSat : entier;  //  k-sat : nombre de littéraux d'une clause
                                    //  nCl : nombre de clause dans la matrice
                                    //  nClSat : nombre de clause satisfaites

Début
  k= getx_SAT(M);
  nCl= getnCl(M);
  nClSat = 0;

    Pour i allant de 1 à nCl faire          // Parcourir toutes les clauses
      Pour j allant de 1 à k_sat  faire     //  Parcourir tous les littéraux de la clause i
          Si ( S [ABS(M[i,j])] = M[i,j] ) alors  nClSat := nClSat + 1;  // Clause satisfaible
                                                 break;                 // sortir du boucle interne
          Fsi;
      Fait;
    Fait;
  retourner(nClSat);
Fin.
```

La complexité de cet algorithme au pire cas (aucune clause n'est satisfaite par la solution en entrée) est égale au produit des extrema des deux boucles « pour » i.e. nombre de colonnes de la matrice multiplier par le nombre de lignes de cette dernière. Donc si on a ‘’n’’ clauses et k littéral dans chacune (k-SAT) alors la complexité est : k*n

IV. Implémentation java :

1- BFS :

La méthode BFS utilise la structure de "file" pour stocker les nœuds crées tout au long de la recherche. Cette file est une liste chaînée où les nouveaux éléments arrivés sont placés à la fin (queue) pendant que l'enlèvement concerne uniquement le premier élément (la tête). Cette file est généralement nommée « Ouvert » et elle signifie que ses nœuds sont des candidats, pas encore traités.

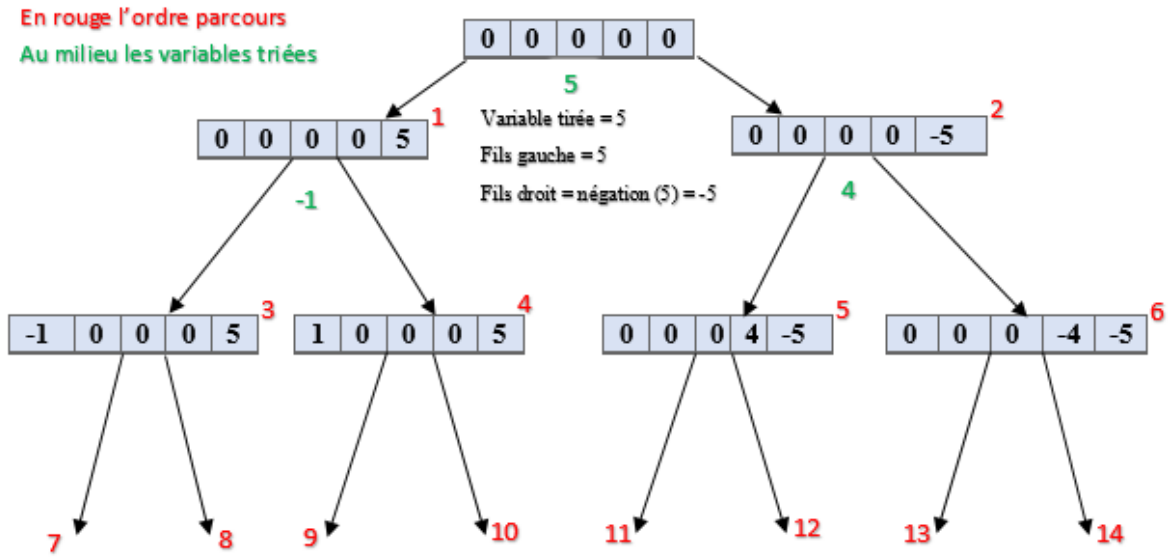
Pour le problème SAT, la modélisation est choisie pour qu'un nœud représente une solution

tout entière, initialement vide, en ajoutant des littéraux, aléatoirement, à chaque passage à un niveau plus profond. Le tirage des littéraux est fait de manière que chaque variable est choisie une seule fois dans la branche courante.

Cette méthode de la recherche nécessite en général une autre liste nommée "fermée" habituellement. Cette liste sauvegarde en mémoire la suite des nœuds développés. « Fermée » est mise à jour lors de l'avancement dans la branche courante en ajoutant un nœud ou lors de changement de branche, donc suppression d'un nœud.

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

On explique l'utilisation des structures choisies pour la recherche en largeur d'abord en montrant un arbre exemple créé en utilisant cette méthode de recherche pour un problème SAT avec 5 variables.



L'étude de la complexité :

On a déjà cité que La recherche en largeur d'abord est très gourmande en espace mémoire. Mais cela dépend d'un côté de l'instance SAT concernée et de l'autre côté, le tirage aléatoire des variables.
Lors de l'étude de la complexité on s'intéresse particulièrement au pire et meilleur cas tels que le pire cas est le cas d'une instance du problème SAT qui n'admet pas de solution i.e. n'est pas satisfiable. Contrairement au meilleur cas qui est le cas d'une instance du problème SAT qui admet une solution avec qu'un seul littéral choisi lors de la première itération.

Donc, on obtient :

✓ **Complexité temporelle au meilleur cas :** une '1' itération afin de trouver la solution. Alors on parle d'une complexité constante égale à O(1).

× **Complexité temporelle au pire cas :** max itérations et à chaque itération, deux nouvelles solutions sont ajoutées à la liste « Ouverte » (sauf pour le dernier niveau). La complexité est alors exponentielle et posant "n" le nombre de variables du problème SAT, cette complexité est égale à $O(2^n)$.

Pour la complexité empirique spatiale, elle est évaluée en fonction de la taille de l'ensemble « Ouverte » en d'autres mots, elle est évaluée en fonction de nombre des nœuds dans cette ensemble :

✓ **Complexité spatiale au meilleur cas :** La solution est atteinte dès la première itération, aucun nœud n'est mis dans la liste « Ouverte ». Donc la complexité est constante, égale à O(1).

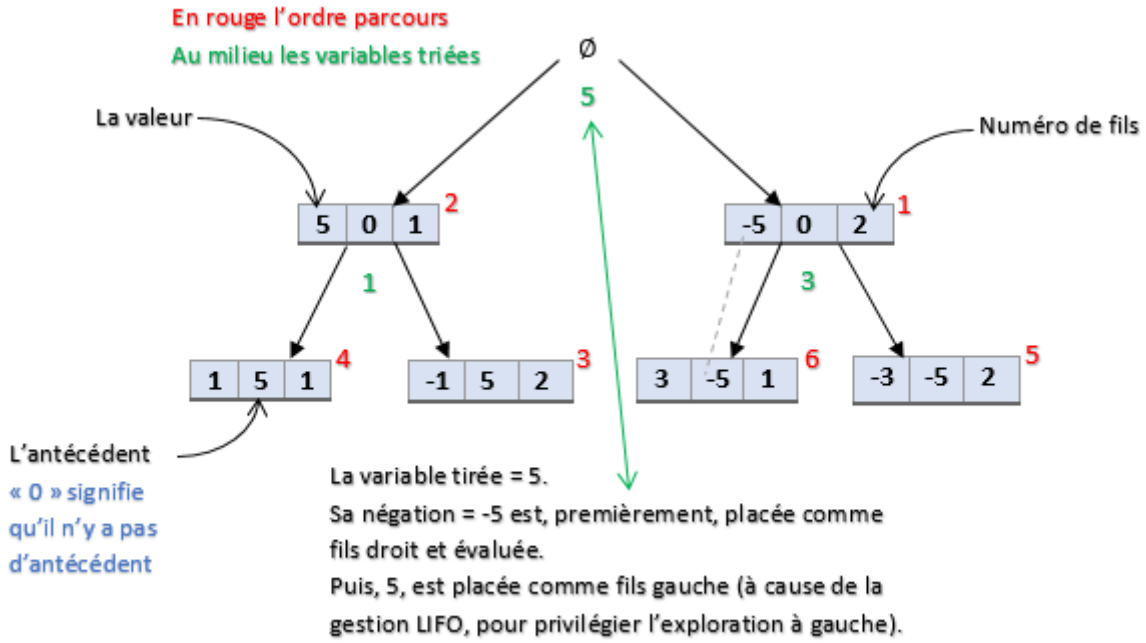
× **Complexité spatiale au pire des cas :** On a deux nouvelles solutions à chaque itération dans l'ensemble « Ouverte » cependant il s'agit aussi de la suppression de l'ancienne solution i.e. le nœud père. Ainsi, la taille maximale de « Ouverte » est atteinte au dernier niveau soit "n". Donc, la complexité est exponentielle et est égale à : $O(2^n)$.

2- DFS :

On a mentionné dans la description de la méthode de la recherche en profondeur d'abord que cette dernière ne mémorise pas tous les nœuds en mémoire, mais uniquement ceux du chemin menant au but. Pour cela elle utilise la structure de donnée « pile » (stack) pour la sauvegarde des nœuds. On dit que la structure de donnée « pile » et très adéquate à la stratégie retour arrière car la tête de la pile est la seule concernée par les ajouts/retraits. De cette manière, le dernier nœud arrivé est susceptible d'être supprimé.
Elle utilise également les 2 listes « Ouverte » et « Fermée » mais leur implémentation est différente car un nœud maintenant ne représente pas une solution potentielle du problème SAT. Il représente un ensemble de uniquement trois entiers ainsi l'économisation du l'espace mémoire utilisé dans la recherche.
Ces trois informations de type **Integer** sont :

- La valeur de la variable (le littéral, positif ou négatif).
- La valeur de la variable du père (l'antécédent).
- Numéro du fils, soit '1' signifiant le 1er fils du l'antécédent (fils gauche) ou bien '2' (2eme fils qui est le fils droit).

L'antécédent et le numéro du fils sont utilisés dans la procédure du retour arrière. C-à-d après l'exploitation finale du premier fils, si la solution n'a pas été trouvée, il faut passer automatiquement au deuxième fils. Cela vaut dire la nécessité de supprimer le premier fils. Et ainsi de suite de façon récursive.



L'étude de la complexité :

On définit la complexité du DFS dans les mêmes deux cas que pour BFS (meilleur cas : solution trouvée à la première itération, pire des cas : solution inexistante).

- ✓ **Complexité temporelle au meilleur cas :** une '1' itération afin de trouver la solution. Alors on parle d'une complexité constante égale à O(1).
- × **Complexité temporelle au pire cas :** max itérations et à chaque itération, deux nouveaux nœuds sont ajoutés à la liste « Ouverte » (sauf pour le dernier niveau). La complexité est alors exponentielle et posant "n" le nombre de variables du problème SAT, cette complexité est égale à $O(2^n)$.

Pour la complexité empirique spatiale, elle est évaluée en fonction de la taille de l'ensemble « Ouverte ».

- ✓ **Complexité spatiale au meilleur cas :** La solution est atteinte dès la première itération, aucun nœud n'est mis dans la liste « Ouverte ». Donc la complexité est constante, égale à O(1).
- × **Complexité spatiale au pire des cas :** Toutes les combinaisons sont essayées. Chaque itération rajoute deux nouveaux nœuds (un de la branche courante, l'autre à traiter ultérieurement) à « Ouverte ». Ainsi, la taille maximale de l'ensemble « Ouverte » est atteinte au dernier niveau (numéro "n") où la branche actuelle est explorée complètement. Alors, la complexité est linéaire, égale à : $2n \rightarrow O(n)$.

3- Algorithme A*

Le A* est une recherche arborescente qui utilise « Ouverte » pour le stockage des nouveaux nœuds créés comme les méthodes aveugles qu'on a vu. La différence est que la liste « Ouverte » est triée en ordre croissant selon la fonction d'évaluation f. Ainsi, le prochain pas d'exploration dépend de l'évaluation heuristique qui vise à estimer le meilleur chemin pour atteindre le but.
Similairement au BFS, un nœud représente une solution complète. La fonction f appliquée à un nœud « n » est calculée en sommant deux mesures heuristiques g et h tel que :

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

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

L'étude de la complexité :

Les cas d'étude de la complexité sont les mêmes qu'auparavant.

✓ **Complexité temporelle au meilleur cas :** une '1' itération afin de trouver la solution. Alors on parle d'une complexité constante égale à O(1).

× **Complexité temporelle au pire cas :** La spécification du nombre d'itérations est impossible car la branche de recherche peut changer plusieurs fois selon l'évaluation. Ainsi, nous considérons que toutes les possibilités seront parcourues par la boucle. La complexité est donc exponentielle, égale à $O(2^n)$ tel que "n" est le nombre de variables du problème SAT.

Pour la complexité empirique spatiale, elle est évaluée en fonction de la taille de l'ensemble « Ouverte » :

✓ **Complexité spatiale au meilleur cas :** La solution est atteinte dès la première itération, aucun nœud n'est mis dans la liste « Ouverte ». Donc On a une complexité constante qui est égale à O(1).

× **Complexité spatiale au pire cas :** Toutes les combinaisons risquent d'être essayées. On dit que la complexité est exponentielle, égale à : $O(2^n)$.

V. Expérimentations

1- Description du Benchmark

Nous allons tester les trois méthodes sur des données de benchmark. Plus précisément, le fichier benchmarks uf75-325. Ce fichier benchmark « uf75-325 » contient des instances satisfiables. Nous allons effectuer nos tests sur seulement les 10 premières instances du fichier ce qui est largement suffisant vu que notre but est d'estimer le taux de satisfiabilité moyen.
Ce benchmark comporte chacun :

- 100 fichiers en format CNF (100 instances de type 3-SAT).
- Chaque fichier contient 325 clauses et 75 variables.
- Chaque clause est composée de 3 variables.

2- Environnement expérimental

L'environnement de travail est le suivant :

RAM: 4Go (3,85Go Utilisable)

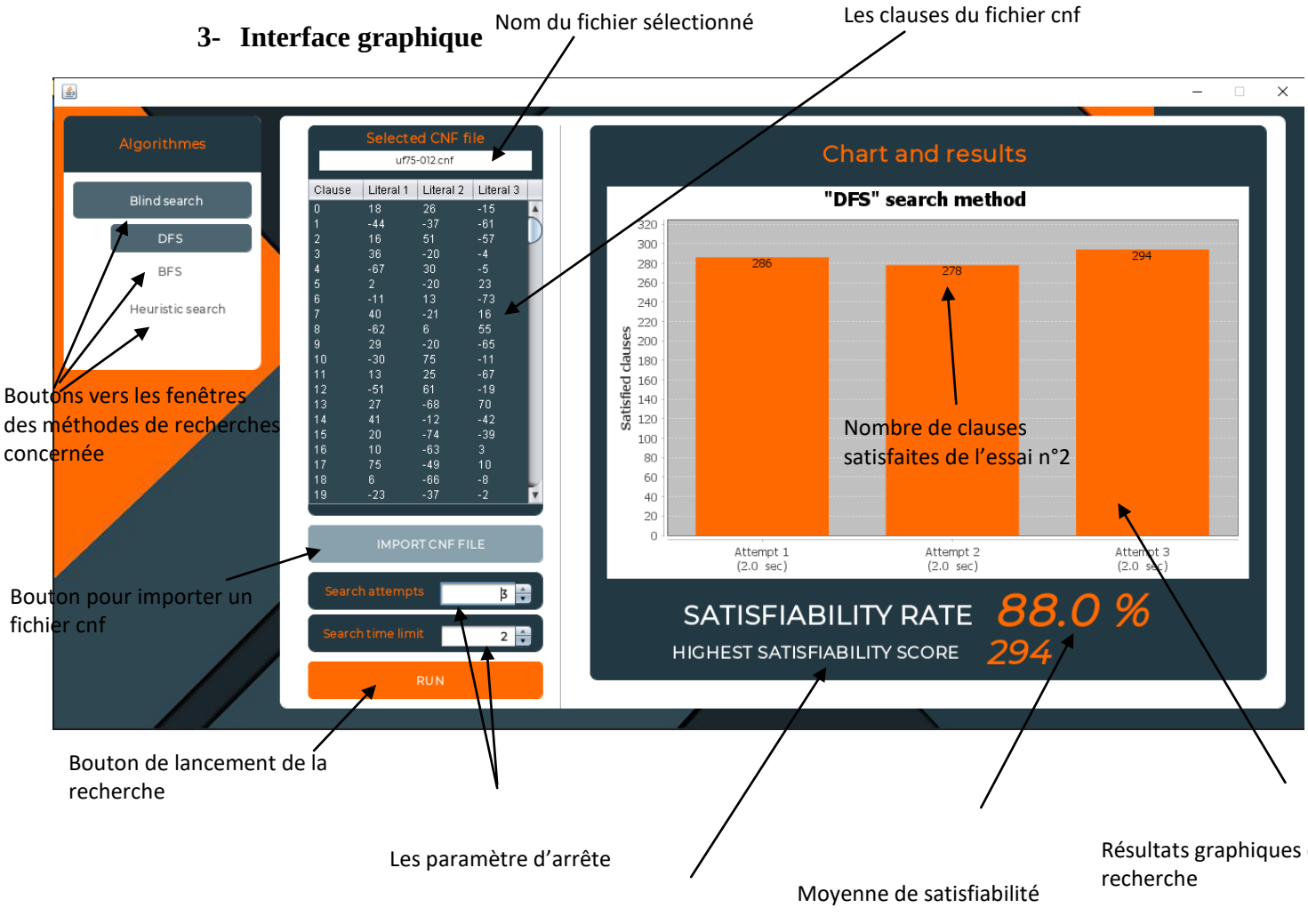
Processeur : Intel® Core™ i3-2328M CPU @ 2.20 GHz 2.20 GHz.

Système d'exploitation : Windows 7.

Langage de programmation : Java.

Environnement de développement : Eclipse.

3- Interface graphique



4- Résultats expérimentaux

Pour les expérimentations ont spécifie les paramètres qui entre en jeux et qui sont seulement ceux d'arrêt. On parle du nombre d'essais ou bien de tentatives pour une instance donnée pour calculer par la suite la moyenne (fiabilité). Et on parle également du temps (seconds) par essai qui est le délai d'exécution d'une tentative. On dit alors qu'une exécution s'arrête si une solution a été trouvée ou son délai est atteint

On englobe les résultats obtenus dans les tableaux suivants :

1- BFS :

Benchmarks	Instance	Nombre d'essais	Temps par essai (en secondes)	Taux moyen de satisfiabilité	Temps moyen d'exécution
UF-75	1	5	3	41.4769%	3
	2	5	3	43.6307%	3
	3	5	3	38.7384%	3
	4	5	3	42.0307%	3
	5	5	3	35,2833%	3
	6	5	3	42.5041%	3
	7	5	3	40.1254%	3
	8	5	3	39,7013%	3
	9	5	3	41.3538%	3
	10	5	3	38,1003%	3

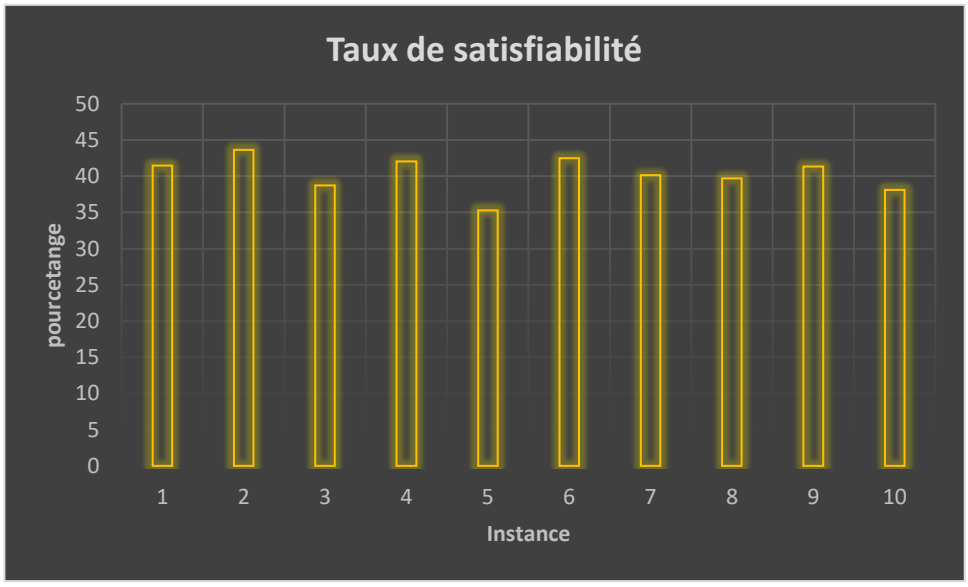


Figure 05 - Histogramme des résultats du test de la méthode BFS sur les fichiers de Benchmark UF-75

2- DFS :

Benchmarks	Instance	Nombre d'essais	Temps par essai (en secondes)	Taux moyen de satisfiabilité	Temps moyen d'exécution
UF-75	1	5	3	88.2516 %	3
	2	5	3	90.7441 %	3
	3	5	3	89.1729 %	3
	4	5	3	89.2431 %	3
	5	5	3	90.2578 %	3
	6	5	3	91.1488 %	3
	7	5	3	89.0011 %	3
	8	5	3	92.0143 %	3
	9	5	3	90.4421 %	3
	10	5	3	88.1398 %	3

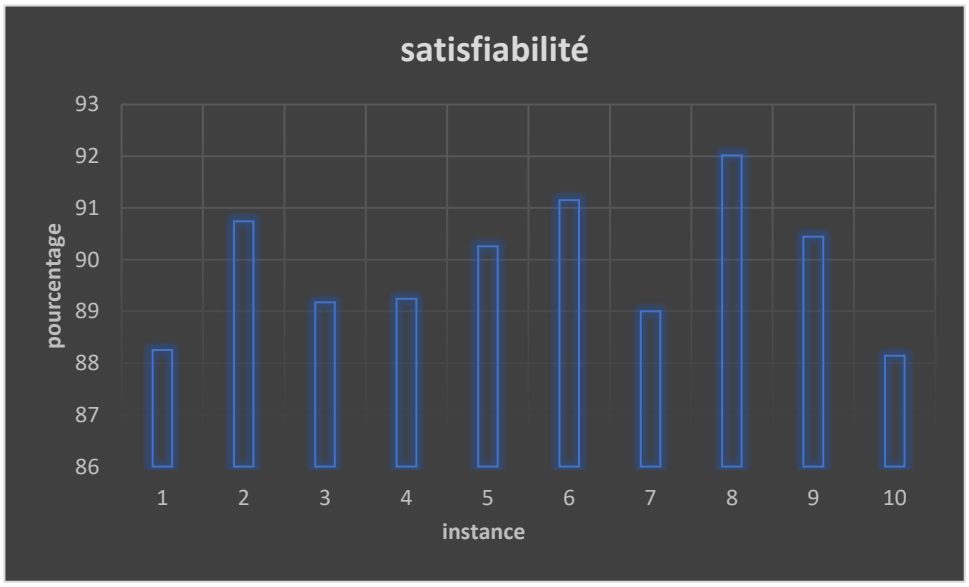


Figure 06 - Histogramme des résultats du test de la méthode DFS sur les fichiers de Benchmark UF-75

3- Algorithme A*

Benchmarks	Instance	Nombre d'essais	Temps par essai (en secondes)	Taux moyen de satisfiabilité	Temps moyen d'exécution
UF-75	1	5	3	92.0615%	3
	2	5	3	94.4782%	2.5
	3	5	3	92.4872%	3
	4	5	3	90.2778%	3
	5	5	3	92.0132%	3
	6	5	3	93.8251%	3
	7	5	3	90.7845%	3
	8	5	3	89.9891%	3
	9	5	3	92.1578%	3
	10	5	3	90.8221%	3

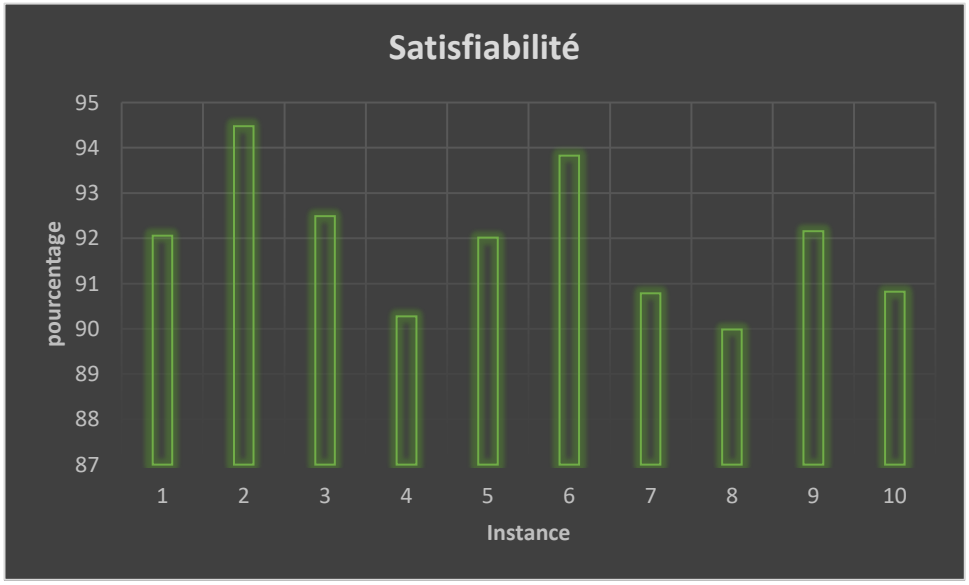
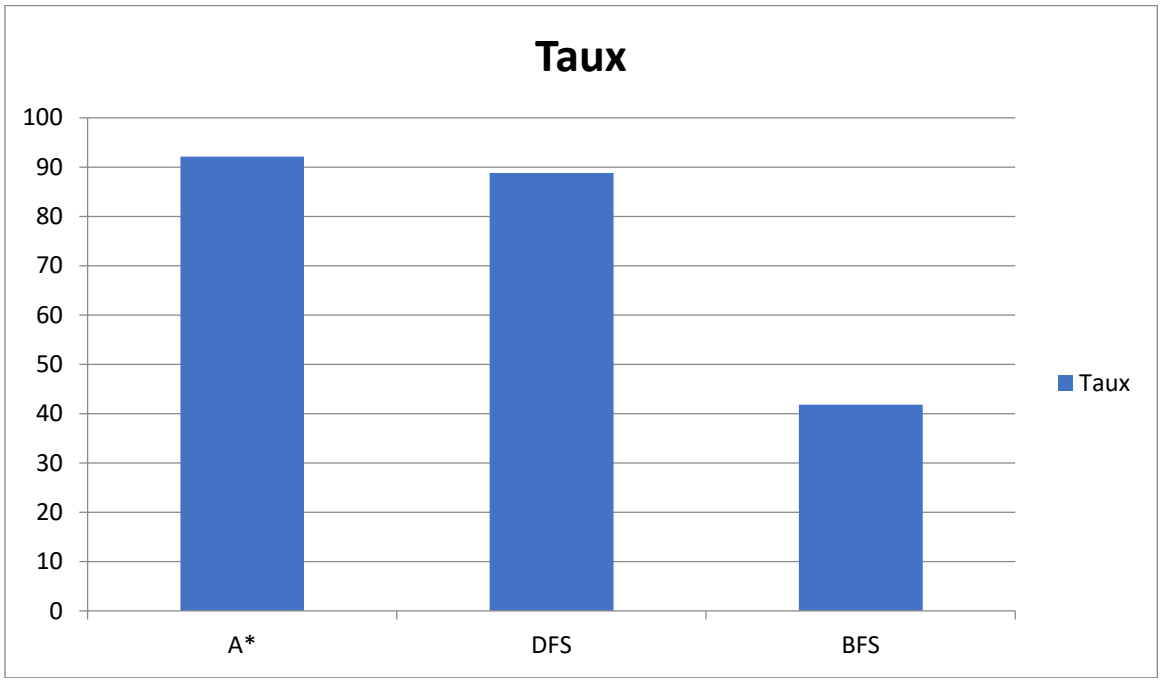


Figure 07 - Histogramme des résultats du test de la méthode heuristique A* sur les fichiers de Benchmark UF-75

5- Etude comparative des 3 méthodes

En analysant ce graphe on remarque bien que la méthode A* qui utilise l'heuristique est mieux que les méthodes BFS et DFS (DFS et BFS donnent presque les mêmes résultats).



VI. Conclusion

Ce projet nous a permis d'approfondir nos connaissances en se familiarisant avec des approches sophistiquées pour résoudre des problèmes complexes. Ces derniers, souvent NP-Complets, n'étant pas solvables avec les techniques classiques (exactes). Pour cela, les méthodes méta-heuristiques sont utilisées cherchant à trouver un compromis entre les ressources utilisées, temps d'exécution et la qualité de la solution. De plus, l'étude approfondie du problème SAT nous a donné la chance de découvrir son importance capitale dans l'étude théorique de la complexité.