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

Université d'Alger 1
Faculté des sciences

Rapport de TP

Résolution de problème de satisfiabilité

Par les étudiantes :
- DEHNI Souhila (G02)
- TIDAFI Asma (G01)

Proposé par :
Dr. DRIAS Yassine

Année universitaire :
2019-2020

Table des matières

1. Introduction :

Le problème de satisfiabilité dit « SAT » est classé comme problème de décision qui étant donné une formule de logique propositionnelle, détermine s'il existe une assignation des variables propositionnelles qui la rende satisfiable (vraie). SAT a été démontré comme un problème NP-complet par Stephan Cook en 1971.

SAT a une grande place dans l'intelligence artificielle. L'étude de ce problème et de ses variantes a permis de pointer sur la frontière entre les problèmes faciles et difficiles. La satisfiabilité des formules logiques fait en sorte qu'on se situe du côté de la résolution de problèmes, du raisonnement automatique et dans tout autre axe de l'IA ou la logique est nécessaire. Il est aujourd'hui utilisé dans de nombreux domaines comme la cryptanalyse, la planification, la bio-informatique, la vérification de matériels et de logiciels…

Depuis plusieurs années, SAT a fait l'objet de nombreux travaux ayant abouti à des résultats remarquables sur le double plan : théorique et pratique. Mais malgré son succès, il existe encore une forte demande d'algorithmes efficaces permettant de résoudre ces problèmes difficiles.

Nous allons dans ce projet tester plusieurs méthodes qui sont soit aveugles ou informés (utilisent une heuristique) ainsi que des algorithmes de recherche évolutionnaires de résolution du problème de satisfiabilité et enfin faire une analyse et comparaison des résultats obtenus.

2. Environnement expérimental :

. RAM : *12GO*
. Processeur : *Intel(R) Core™ i5-6200U CPU @ 2.30 GHz 2.40 GHz.*
. Système d'exploitation : *Windows 10-64 bits*
. Langage de programmation : *JAVA*
. Environnement de développement : *Eclipse.*

3. Lecture et traduction du fichier :

3.1. Principe :

Les fichiers des instances sont dans le format CNF et se présente comme dans l'exemple qui suit :

```
c This Formular is generated by mcnf
c
c    horn? no
c    forced? no
c    mixed sat? no
c    clause length = 3
c
p cnf 75  325
 42 22 15 0
73 -22 -24 0
50 -20 53 0
-66 49 -15 0
39 8 -74 0
66 72 -61 0
-32 -13 -8 0
```

La ligne qui commence par un « p » contient le nombre de variables de l'instance (75 dans l'exemple) et le nombre de clauses (325 dans l'exemple).

Toutes les clauses ont la même taille. Si le littéral est négatif il est présenté par le nombre de la variable signé négative (précédée par un « - ». S'il est positif il est présenté par le nombre de la variable seulement. On ne retrouve que les variables existantes dans la clause.

Pour la traduction et simplification de ce fichier, nous avons créé une fonction qui nous retourne en sortie une matrice d'adjacence qui a le nombre de variables comme nombre de lignes et le nombre de clauses comme nombre de colonnes :

Si la variable est négative : 0 ;
Si la variable est positive : 1 ;
Si la variable n'existe pas dans la clause : -1.

3.2. La fonction *« parse »* :

```
public static int[][] parse(InputStream inputstream){
     int nbVar;
     int nbCls;
     int i = 0;
     int str;

     //déclarer un scanner pour parcourir le contenu du fichier
     @SuppressWarnings("resource")
     Scanner scanner = new Scanner(inputstream);

     //dépasser les lignes qui commencent par le caractère 'c'
     String token = scanner.next();
     while (token.equals("c")) {
         scanner.nextLine();
         token = scanner.next();
     }

     //Lire la tête du fichier et récuperer le nombre de variables et le nombre de clauses
     scanner.next();
     nbVar = scanner.nextInt();
     nbCls = scanner.nextInt();

     //créer et initialiser la matrice résultante de clauses à -1
     int[][] clauses = new int[nbCls][nbVar];
     for (int[] row: clauses) Arrays.fill(row, -1);

     //lire les clauses ligne par ligne et les codifier
     //-1 si la variable n'existe pas
     //1 si la variable est positive
     //0 si la variable est négative
     while(i<nbCls) {
         while((str = scanner.nextInt()) != 0){
             if(str < 0) {
                 clauses[i][-str-1] = 0;
             }
             else {
                 clauses[i][str-1] = 1;
             }
         }
         i++;
         scanner.hasNextLine();
     }
     return clauses;
}
```

4. Partie 1 :

Pour la recherche en largeur et la recherche en profondeur, nous avons utilisé la même structure de données qui est un arbre binaire.

Nous avons commencé par la création de la classe *« Node »* qui présente un nœud d'un arbre :

```java
public class Node {
    int value;
    Node left;
    Node right;
    Node pere;
    int niveau;

    public int getValue() {

    public void setValue(int value) {

    public Node getLeft() {

    public void setLeft(Node left) {

    public Node getRight() {

    public void setRight(Node right) {

    public Node getPere() {

    public void setPere(Node pere) {

    public int getNiveau() {
        return niveau;
    }

    public void setNiveau(int niveau) {

    Node(int value) {
}
```

Ensuite nous avons créé la classe « *BinaryTree* » qui représente l'arbre binaire :

```java
public class BinaryTree {
    Node root;

    public BinaryTree(int value) {
        root = new Node(value);
    }

    public void addLeft(Node node) {
        node.left = new Node(0);
    }

    public void addRight(Node node) {
        node.right = new Node(1);
    }

    public Node getRoot() {
        return root;
    }

    public void setRoot(Node root) {
        this.root = root;
    }
}
```

Les fonctions « *addLeft* » et « *addRight* » ajoutent un nœud avec valeur statique (0 et 1 respc.) à l'arbre binaire.

4.1. Recherche en profondeur d'abord :

1- Codage de la solution :

Pile : Contient les nœuds à visiter.

Arbre binaire : Contient le chaînage des solutions.

Tableau : Contient la solution (sa taille = nombre de variables).

2- Pseudo code :

Algorithme DFS
Entrée: s: état initial; F: ensemble des états finaux; seuil: entier;
Sortie: une solution (chaîne) si succès sinon échec;
Var: Ouverte: pile de nœuds initialement vide; Fermée: file de nœuds initialement vide;
Début

1. empiler le nœud initial *s* dans la pile *Ouverte* ;
2. **si** (*Ouverte* = {∅}) **alors** **échec** **sinon** continuer;
3. dépiler *n*, le premier nœud de la pile *Ouverte* et l'enfiler dans *Fermée*;
4. **si** *n* n'a pas de successeur **ou** la profondeur de l'arbre est égale au *seuil* **alors** aller à 2;
5. déterminer les successeurs de *n* et les empiler dans *Ouverte;*
6. créer un chainage de ces nœuds vers *n*;
7. **si** parmi les successeurs il existe un état final **alors** **succès**: la solution est la chaîne des nœuds allant du courant à la racine;
8. **sinon** aller à 2;

Fin

3- Fonction *« DepthFirst »* :

```
public class DepthFirst {
    public static int[] depth(BinaryTree bt, int[][] clauses) throws FileNotFoundException {
        int g;

        //nombre de variables et de clauses de l'instance
        int nbVar = clauses[0].length;
        int nbCls = clauses.length;

        //pointer sur la racine
        Node node = bt.getRoot();
        node.setPere(node);
        node.setNiveau(0);

        //la pile qui nous permet de revenir au noeud ou on s'est arreté
        LinkedList<Node> pile = new LinkedList<Node>();

        //on ajout la racine à la pile
        pile.add(node);

        //sol est le tableau des solutions
        int[] sol = new int[nbVar];
        Arrays.fill(sol, -1);

        while(!pile.isEmpty()) {
            if(node.getLeft() == null) {
                while(node.getNiveau() < nbVar) {
                    //ajouter le fils gauche, son niveau (actuel + 1) et son pere
                    bt.addLeft(node);
                    node.getLeft().setNiveau(node.getNiveau()+1);
                    node.getLeft().setPere(node);

                    //récupérer la solution à partir de la racine
                    Test.getSolFromRoot(node.getLeft(),sol, node.getNiveau());

                    //calculer combien de clauses satisfait cette solution
                    g = Test.clausesSat(clauses, nbCls, node.getLeft().getNiveau(), sol);

                    //tester si la bonne solution, si c'est le cas la renvoyer
                    if(g == nbCls) {
                        return sol;
                    }

                    node = node.getLeft();


                    //si nous avons pas encore atteint de dernier niveau de l'arbre, on ajoute la noeud à la pile
                    if(node.getNiveau() < nbVar) {
                        pile.addLast(node);
                    }
                }
            }
            //dépiler le dernier noeud ajouté
            node = pile.removeLast();

            //ajouter le fils droit, son niveau (actuel + 1) et son pere
            bt.addRight(node);
            node.getRight().setNiveau(node.getNiveau()+1);
            node.getRight().setPere(node);

            //récupérer la solution à partir de la racine
            Test.getSolFromRoot(node.getRight(),sol, node.getNiveau());

            //calculer combien de clauses satisfait cette solution
            g = Test.clausesSat(clauses, nbCls, node.getLeft().getNiveau(), sol);

            //tester si la bonne solution, si c'est le cas la renvoyer
            if(g == nbCls) {
                return sol;
            }

            //si nous avons pas encore atteint de dernier niveau de l'arbre, on ajoute la noeud à la pile
            if(node.getRight().getNiveau() < nbVar) {
                pile.addLast(node.getRight());
                node = pile.peekLast();
            }

        }
        return null;
    }
}
```

4- **Complexité temporelle :** $O(2^{75})$

5- **Complexité spatiale :** $O(2^{75})$

Avec 75 : La profondeur de l'arbre (qui est le nombre de variables).

4.2. Recherche en largeur d'abord :

1- Codage de la solution :

File : Contient les nœuds à visiter.

Arbre binaire : Contient le chaînage des solutions.

Tableau : Contient la solution (sa taille = nombre de variables).

2- Pseudo code :

Algorithme BFS
Entrée: s: état initial; F: ensemble des états finaux;
Sortie: une solution (chaîne) si succès sinon échec;
Var: Ouverte, Fermée: file de nœuds initialement vides;
Début
1. insérer le nœud initial *s* dans la file *Ouverte* ;
2. **si** (*Ouverte* = {∅}) **alors** échec **sinon** continuer;
3. défiler *n*, le premier nœud de la file *Ouverte* et l'insérer dans *Fermée*;
4. **si** *n* n'a pas de successeur **alors** aller à 2;
5. déterminer les successeurs de *n* et les enfiler dans *Ouverte;*
6. créer un chainage de ces nœuds vers *n*;
7. **si** parmi les successeurs il existe un état final **alors** succès: la solution est la chaîne des nœuds allant du courant à la racine;
8. **sinon** aller à 2;

Fin

3- Fonction *« BreadthFirst »*

```
public class BreadthFirst {
    //arbre binaire pour créer le chainage
    public static int[] breadth(BinaryTree bt, int[][] clauses) throws FileNotFoundException {
        int g;

        //nombre de variables et de clauses
        int nbVar = clauses[0].length;
        int nbCls = clauses.length;

        int niveau = 0;
        int extaille = 1;
        int taille = 0;

        //pointer sur la racine de l'arbre
        Node node = bt.getRoot();

        //la pile qui nous permet de revenir au noeud ou on s'est arreté
        LinkedList<Node> file = new LinkedList<Node>();

        //on ajoute la racine à la pile
        file.add(node);

        //sol est le tableau des solutions
        int[] sol = new int[nbVar];
        Arrays.fill(sol, -1);

        while(!file.isEmpty()) {
            //pointer sur le premier noeud
            node = file.removeFirst();

            //travailler le fils gauche et droit du noeud et les insérer dans la file

            //ajouter le fils gauche et son pere
            bt.addLeft(node);
            node.getLeft().setPere(node);

            taille++;

            //ajouter le noeud à la file
            file.addLast(node.getLeft());

            //récupérer la solution à partir de la racine
            Test.getSolFromRoot(node.getLeft(),sol, niveau);

            //calculer combien de clauses satisfait cette solution
            g = Test.clausesSat(clauses, nbCls, niveau+1, sol);

            //tester si la bonne solution, si c'est le cas la renvoyer
            if(g == nbCls) {
                return sol;
            }

            //-----------------------------------------------------------

            //ajouter le fils droit et son pere
            bt.addRight(node);
            node.getRight().setPere(node);

            taille++;

            //ajouter le noeud à la file
            file.addLast(node.getRight());

            //récupérer la solution à partir de la racine
            Test.getSolFromRoot(node.getRight(),sol, niveau);

            //calculer combien de clauses satisfait cette solution
            g = Test.clausesSat(clauses, nbCls, niveau+1, sol);

            //tester si la bonne solution, si c'est le cas la renvoyer
            if(g == nbCls) {
                return sol;
            }

            if(taille == extaille*2) {
                niveau++;
                extaille = taille;
```

```
            if(taille == extaille*2) {
                niveau++;
                extaille = taille;
                taille = 0;
            }
        }

        //s'il ne y a pas de solution on ne retourne rien
        return null;
    }
}
```

6- **Complexité temporelle :** $O(2^{75})$

7- **Complexité spatiale :** $O(2^{75})$

Avec 75 : La profondeur de l'arbre (qui est le nombre de variables).

4.3. Recherche A* :

1- **Objectif :** Maximiser la fonction fitness.

2- **La fonction fitness :** $f(x) = g(x) + h(x)$;

h(xi) : Heuristique qui est dans ce cas le nombre de clauses qui contiennent la variable *xi ;*

g(xi) : Le coût qui est dans ce cas le nombre de clauses que satisfait la solution actuelle.

3- **Pseudo code :**

Algorithme A*
Entrée: s: état initial; F: ensemble des états finaux;
c(n_i, n_j): le coût entre les nœuds adjacents n_i et n_j; h: fonction heuristique;
Sortie: une solution (chaîne) si succès sinon échec;
Var: Ouverte : liste de nœuds triée selon la fonction f et initialement vide;
Fermée: file de nœuds initialement vide;
Début
1. insérer le nœud initial s dans la liste *Ouverte* avec $f(s) := g(s) + h(s)$;
2. **si** (*Ouverte* = {∅}) **alors échec sinon** continuer;
3. retirer n, le premier nœud de la liste *Ouverte* et l'enfiler dans *Fermée*;
4. **si** n n'a pas de successeur **alors** aller à 2;
5. **Pour chaque** successeur n_i de n
6. **Faire** $f(n_i) := g(n_i) + h(n_i)$;
7. | insérer n_i avec $f(n_i)$ dans *Ouverte* selon l'ordre croissant de f;
8. **Fait** créer un chainage de n_i vers n;
9. **si** parmi les successeurs il existe un état final **alors succès**: la solution est la chaîne des nœuds allant du courant à la racine; **sinon** aller à 2;

Fin

4- Codage de la solution :

Tableau : Contient la solution (sa taille = nombre de variables).

Liste « *Open* » : Contient les nœuds qui n'ont pas encore été visités.

Liste « *Closed* » : Contient les nœuds déjà visités.

Liste de type « *Liste* » pour créer le chaînage. Chaque nœud Liste contient une liste telle que tous les nœuds développés à partie de ce nœud soient stockés dedans.

5- La classe « *Liste* » :

```
public class List {
    int val;
    int nb;
    int h;
    List pere;
    LinkedList<List> list;

    List(int val, int nb) {
        this.val = val;
        this.nb = nb;
    }
}
```

6- La classe « *AStar* » :

Cette classe contient les méthodes suivantes :

a. *checkIfExists()* : Une méthode qui vérifie si un nœud existe dans une liste de listes.

```
public static boolean checkIfExists(LinkedList<List> closed, int nb, int val) {
    List p;
    @SuppressWarnings("unchecked")
    LinkedList<List> l = (LinkedList<List>) closed.clone();
    while(!l.isEmpty()) {
        p = l.remove();
        if(p.val == val && p.nb == nb) {
            return true;
        }
    }
    return false;
}
```

b. *addSort()* : Une méthode qui ajoute un noeud dans une liste telle sorte que cette dernière soit toujours triée dans l'ordre décroissant.

```
public static LinkedList<List> addSort(LinkedList<List> l, List n) {
    if(l.isEmpty()) {
        l.addFirst(n);
    }
    else if(n.h >= l.peekFirst().h) {
        l.addFirst(n);
    }
    else if(n.h <= l.peekLast().h) {
        l.addLast(n);
    } else {
        List f = l.peekFirst();
        while(n.h < l.peekFirst().h) {
            l.addLast(l.removeFirst());
        }
        l.addLast(n);
        while(l.peekFirst() != f) {
            l.addLast(l.removeFirst());
        }
    }
    return l;
}
```

c. *getSolFromRoot()* : Une méthode qui nous retourne la solution d'un noeud depuis la racine.

```
//méthode qui nous retourne la solution d'un noeud depuis la racine
public static void getSolFromRoot(List n, int[] sol, LinkedList<List> closed) {
    List f = n;
    closed.add(f);
    closed.removeAll(closed);
    sol[f.nb - 1] = f.val;
    while(f.pere != null) {
        sol[f.pere.nb - 1] = f.pere.val;
        f = f.pere;
        closed.add(f);
    }
}
```

d. *fitness()* : Une méthode qui calcule la valeur fitness de la solution.

```
//méthode qui calcule la fitness
public static int fitness(int[][] clauses, int[] sol, int i) {
    int g = 0;
    int gi;
    int h = 0;
    int nbCls = clauses.length;
    int nbVar = clauses[0].length;

    for(int l=0; l<nbCls; l++) {
        gi = 0;
        h = 0;
        for(int k=0; k<nbVar; k++) {
            if(clauses[l][k] == sol[k] && sol[k] != -1) {
                gi = 1;
                break;
            }
        }

        for(int k=0; k<nbVar; k++) {
            if(sol[k] != -1) {
                if(clauses[l][k] != -1) {
                    h = 1;
                } else {
                    h = 0;
                    break;
                }
            }
        }
        g = g + gi + h;
    }
    return g;
}
```

e. *AStar()* : Qui est la méthode principale.

```
public static int[] Astar(int[][] clauses) {
    int g;

    int nbVar = clauses[0].length;
    int nbCls = clauses.length;

    //initialisation du tableau de solution
    int[] sol = new int[nbVar];
    Arrays.fill(sol,-1);

    //les listes open et closed
    LinkedList<List> open = new LinkedList<List>();
    LinkedList<List> closed = new LinkedList<List>();

    //liste de listes pour le chainage
    List l = new List(-1,0);

    //ajouter la racine à la liste open
    open.add(l);

    //un pointeur sur le noeud courant
    List current;

    //tant qu'il y a toujours des noeuds à parcourir dans la liste open on rentre dans la boucle
    while(!open.isEmpty()) {

        //pointer sur la tête de la liste open, ie. le noeud avec la plus grande fitness
        current = open.removeFirst();

        if(current.nb != 0) {
            if(sol[current.nb -1] != -1) {
                Arrays.fill(sol,-1);
                getSolFromRoot(current,sol,closed);
            }
        }

        //ajouter le noeud qu'on va parcourir dans la liste des noeuds visités
        closed.addLast(current);


         //creer une liste qui pointe sur la liste de chainage
         LinkedList<List> q = new LinkedList<List>();

         //pout chaque variable, tester pour les cas 0 et 1
         for(int i=0; i<nbVar; i++) {
            //si le noeud existe déja dans la liste closed, ie. il a déjà été parcouru on ne l'ajoute par dans open
            //si c'est la même variable on ne l'ajoute pas à open car elle a déjà une valeur
            if((!checkIfExists(closed,i+1,0) && !checkIfExists(closed,i+1,1))&& current.nb != i+1) {
                //créer le chainnage
                q.addLast(new List(0,i+1));

                //insérer la solution de la variable pour le test
                sol[i] = 0;

                //calculer la valeur de fitness correspondante au noeud
                q.peekLast().h = fitness(clauses, sol, i);

                //définir le père du noeud
                if(current.val == -1) q.peekLast().pere = null;
                else q.peekLast().pere = current;

                //ajouter le noeud dans open en gardant la liste triée
                open = addSort(open,q.peekLast());

                //apres le calcul de fitness on remet la valeur de cette variable à -1 (on ne sait pas encore quelle
                sol[i] = -1;
            }

            //-------------------------------------------------------------


            if((!checkIfExists(closed,i+1,1) && !checkIfExists(closed,i+1,0))&& current.nb != i+1) {
                q.addLast(new List(1,i+1));

                sol[i] = 1;

                q.peekLast().h = fitness(clauses, sol, i);

                if(current.val == -1) q.peekLast().pere = null;
                else q.peekLast().pere = current;
```

```
            if((!checkIfExists(closed,i+1,1) && !checkIfExists(closed,i+1,0))&& current.nb != i+1) {
                q.addLast(new List(1,i+1));

                sol[i] = 1;

                q.peekLast().h = fitness(clauses, sol, i);

                if(current.val == -1) q.peekLast().pere = null;
                else q.peekLast().pere = current;

                open = addSort(open,q.peekLast());

                sol[i] = -1;
            }
        }

        //construction de la solution
        if(!open.isEmpty()) {
            Arrays.fill(sol,-1);
            getSolFromRoot(open.peekFirst(),sol,closed);
        }
        for(int ll=0; ll<sol.length; ll++) {
            System.out.print(sol[ll]+" ");
        }
        System.out.println();

        //calculer combien de clause cette solution satisfait
        g = Test.clausesSat(clauses,nbCls,nbVar,sol);
        System.out.println("Le nombre de clauses satisfaites: "+g);

        //vérifier si c'est la bonne solution et la renvoyer si c'est le cas
        if(g == nbCls) {
            System.out.println("TOUTES LES CLAUSES SONT SAISFAITES: "+g);
            return sol;
        }

        q = null;
    }


    //s'il ne y a pas de solution on ne retourne rien
    return null;
```

8- Complexité temporelle : $O(2^{75})$

9- Complexité spatiale : $O(2^{75})$

Avec 75 : La profondeur de l'arbre (qui est le nombre de variables).

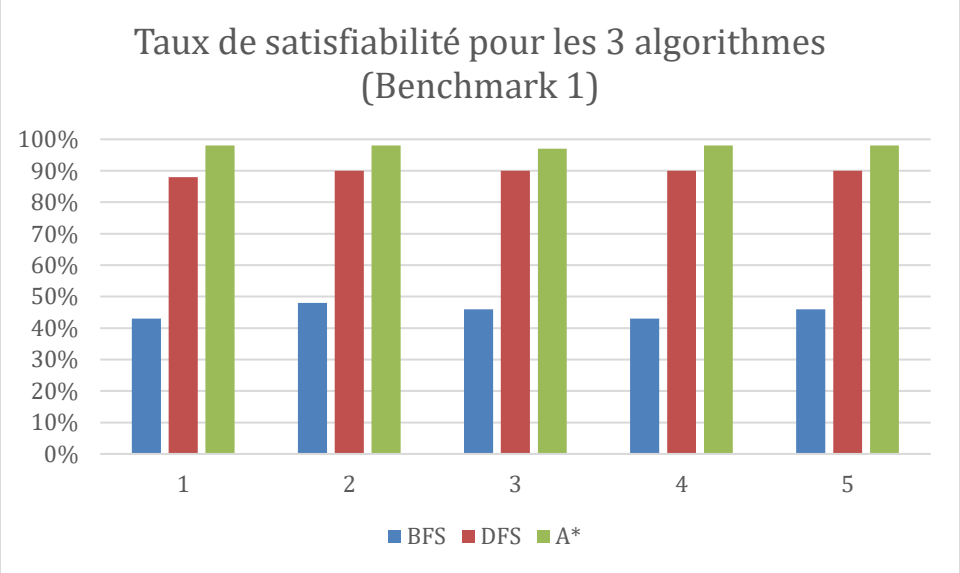
4.4. Analyse et comparaison des résultats :

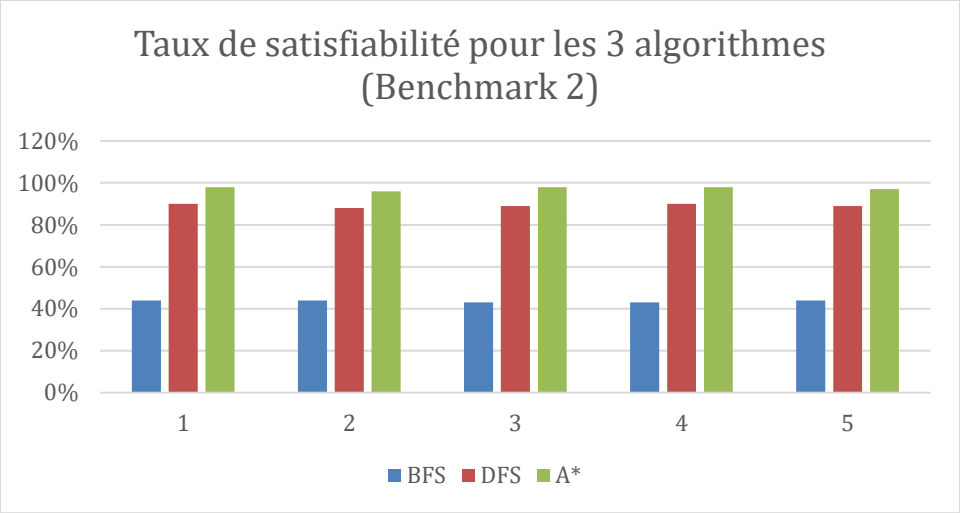
NCS : Nombre de clauses satisfaites au bout de 15 mins.

Benchmarks	Instance	NCS			%		
		BFS	DFS	A*	BFS	DFS	A*
UF75-325	1	141	285	320	43%	88%	98%
	2	156	291	320	48%	90%	98%
	3	149	293	314	46%	90%	97%
	4	140	291	317	43%	90%	98%
	5	150	290	319	46%	90%	98%
UUF75-325	1	144	291	319	44%	90%	98%
	2	144	287	312	44%	88%	96%
	3	142	290	317	43%	89%	98%
	4	140	291	318	43%	90%	98%
	5	143	288	315	44%	89%	97%

Le temps moyen d'exécution des trois algorithmes est : >60mins.

L'algorithme de recherche en largeur d'abord s'arrête au bout de 25mins et donc ne donne pas de résultat.





4.5. Conclusion :

Après observation et analyse des résultats obtenus des tests des trois algorithmes de résolutions sur quelques benchmarks, on peut conclure qu'aucun algorithme n'a pu atteindre le but au bout de 15 mins ou moins.

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 de recherche.

Il faut noter que le pourcentage d'échec n'est pas uniquement lié à la méthode utilisée, mais aussi à la performance de la machine.

5. Partie 2 :

5.1. Algorithme génétique :

1- Codage de la solution :

Tableau de chromosome : Contient la solution (sa taille = nombre de variables).

Tableau de population: Contient ensemble de solutions (sa taille = la taille de la population).

2- Pseudo code :

Input: $Population_{size}$, $Problem_{size}$, $P_{crossover}$, $P_{mutation}$
Output: S_{Gbest}

```
Population := InitializePopulation(Population_size, Problem_size )
EvaluatePopulation(Population)
S_Gbest := GetBestSolution(Population)
While (not StopCondition())
    Parents := SelectParents(Population, Population_size)
    Children := {∅}
    For (Parent_1, Parent_2 ϵ Parents)
        Child_1 := Crossover(Parent_1, Parent_2, P_crossover)
        Child_2 := Crossover(Parent_2, Parent_1, P_crossover)
        Children := Mutate(Child_1, P_mutation)
        Children := Mutate(Child_2, P_mutation)
    End
    EvaluatePopulation(Children)
    S_best := GetBestSolution(Children)
    if (S_best > S_Gbest ) then S_gbest := S_best
    Population := Replace(Population, Children)
End
Return (S_best)
```

3- La classe « Chromosome » :

Cette classe représente une solution de la population :

```
public class Chromosome {
    int [] gene;
    int fitness;
```

Dans la même classe on a rajouté deux fonctions :

a. *Mutate()* : qui fait la mutation dans un chromosome :

```
void mutate(double mutationRate) {
          for(int i = 0; i < gene.length; i++)
           if ( new Random().nextDouble()<= mutationRate) {
               if (gene[i]==1) gene[i] = 0;
               else {if (gene[i]==0) gene[i] = 1;}
                  }
   }
```

La fonction de mutation génère pour chaque variable une valeur aléatoire avec un objet de la classe « *Random* », si elle est inférieure à la valeur *mutationRate*, une mutation est appliquée sur ce bit, sinon elle n'est pas faite, et ceci est appliqué sur tout le chromosome.

b. *calculeFitness()* : Qui évalue le chromosomes. La valeur de la fonction objective « *Fitness* » est égal au nombre de clauses satisfaites.

```
public int calculefitness(int [][]tab,int nbClause,int nbVar) {
    int b=0;
    int fitness=0;
    for(int j=0;j<nbClause;j++)
    {b=0;
        for(int i=0;i<nbVar;i++)
        {
            if(((tab[j][i]==1)&&(this.gene[i]==1)) || ((tab[j][i]==0)&&(this.gene[i]==0)) )
            {fitness++; b=1;break;}
            if (b==1) break;
        }

    }  this.fitness=fitness;
    return fitness;
```

4- La classe « *Population* » :

La classe « *Population* » contient un tableau de solutions de la population i.e. une génération.

```
public class Population {

    Chromosome [] individu;
    int sizePopulation;
```

Dans la même classe nous avons défini deux fonctions :

a. *evalutePopulation()* : Evalue la population. Pour chaque chromosome on fait appel à la fonction *calculeFitness()*.

```
public void evaluatPopulation(int[][]tab, int nbClause,int nbVar) {

        for (int i=0;i<sizePopulation;i++)
        {individu[i].calculefitness(tab, nbClause, nbVar);
                    }
}
```

b. *getSBest()* : Extraire le chromosome qui la valeur maximum de fitness.

```
public int  getSGBest(Population p) {
    int max=0;
    int pos=0;
    for (int i = 0; i < p.sizePopulation; ++i) {
        if (p.individu[i].fitness > max) {
            max = p.individu[i].fitness;
            pos=i;
        }
    }

    return pos;
}
```

5- La classe « *Genetic* » :

La classe principale contient les méthodes suivantes :

a. *tri()* : La fonction qui trier les individus par rapport à leurs valeurs de fitness.

```
public static void tri(Population p) {
    int longueur = p.sizePopulation;
    Chromosome tampon;
    boolean permut;
    int cpt=0;

    do {
        // hypothèse : le tableau est trié
        permut = false;
        for (int i = 0; i < longueur - 1; i++) {
            // Teste si 2 éléments successifs sont dans le bon ordre ou non
            if (p.individu[i].fitness < p.individu[i+1].fitness) {
                // s'ils ne le sont pas, on échange leurs positions
                tampon = p.individu[i];
                p.individu[i] = p.individu[i+1];
                p.individu[i+1]= tampon;
                permut = true;
            }
        }
        } while (permut);//cpt<longueur
    }
```

b. *SelectionParent()* : La fonction qui sélectionne les parents selon le nombre de parents. Tout d'abord la population est triée, ensuite les *nbParentSelectionne* premiers sont récupérés.

```
static Chromosome[] SelectionParent(Population population,int nbParentSelectionne) {
Chromosome [] selection = new Chromosome[nbParentSelectionne];
///trier le tableau des individu par raport a la fitness
tri(population);
  for (int i = 0; i < nbParentSelectionne; i++) {
    selection[i]=population.individu[i];
  }
return selection;
}
```

c. « *singleCrossover()* » : Fait le croisement à point unique, ce dernier est envoyé en paramètre : *pcrossover*.

```
public static  Chromosome singleCrossover(Chromosome parent1,Chromosome parent2,Chromosome children,int pcrossover1,int nbVar){

          for (int i = 0; i <pcrossover1; i++)  children.gene[i]=parent1.gene[i];
          for (int i = pcrossover1; i <nbVar; i++)  children.gene[i]=parent2.gene[i];


  return children;
 }
```

d. « uniformCrossover() ».

```
Chromosome uniformCrossover(Chromosome parent1,Chromosome parent2,Chromosome children,int pcrossover,int nbVar) {

    Random random = new Random();
   for (int i = 0; i < nbVar; i++) {
   if ( (Math.random()*2)<=1) {
       children.gene[i]=parent1.gene[i];
   } else {
       children.gene[i]=parent2.gene[i];
   }
   }
    return children;
}
```

6- La classe « Main » :

En premier, nous avons initialisé notre population, ainsi que les paramètres empiriques. Ensuite, nous avons évalué la population et avons extrait le « *gBest* ». On rentre ensuite dans la boucle et on itère pour *nbrGeneration*.

En fin, on a réévalué l'ancienne population et les nouveaux enfants. Ces derniers sont triés pour ensuite en prendre les premiers *populationSize* pour la prochaine solution.

Tout en évaluant à chaque itération le *sBest* qui est comparé au *gBest*.

Le *gBest* est retournée à la fin de l'algorithme.

```
for (int generationIndex = 0; generationIndex < nbrgeneration; generationIndex++) {
    Population tmp=new Population(populationSize+nbParentSelectionne,sizeproblem);

    for (int j=0;j<populationSize;j++)
      {for (int i=0;i<nbVar;i++)
  {
        tmp.individu[j].gene[i]=population.individu[j].gene[i];
  } }

    Chromosome children1 = new Chromosome(nbVar);
    Chromosome children2 = new Chromosome(nbVar);
    Chromosome[] potentialParents = SelectionParent(population,nbParentSelectionne);
    a=0;compte=0;
    Population childrens = new Population(nbParentSelectionne,sizeproblem);
    while(a<nbParentSelectionne)
    { Chromosome parent1 = potentialParents[a];  a=a+1;
      Chromosome parent2 = potentialParents[a];a=a+1;
        if (new Random().nextDouble() <= pcrossover) {
        children1=singleCrossover( parent1, parent2, children1, pointcrossover,nbVar);
        children2=singleCrossover( parent2, parent1, children2, pointcrossover,nbVar);
            }
        else {
            children1=parent1;
            children2=parent2;
             }
        children1.mutate(mutationRate);
        children2.mutate(mutationRate);
        children1.calculefitness(clauses, nbClause, nbVar);
        children2.calculefitness(clauses, nbClause, nbVar);
```

7- Complexité temporelle : $O(2^{75})$

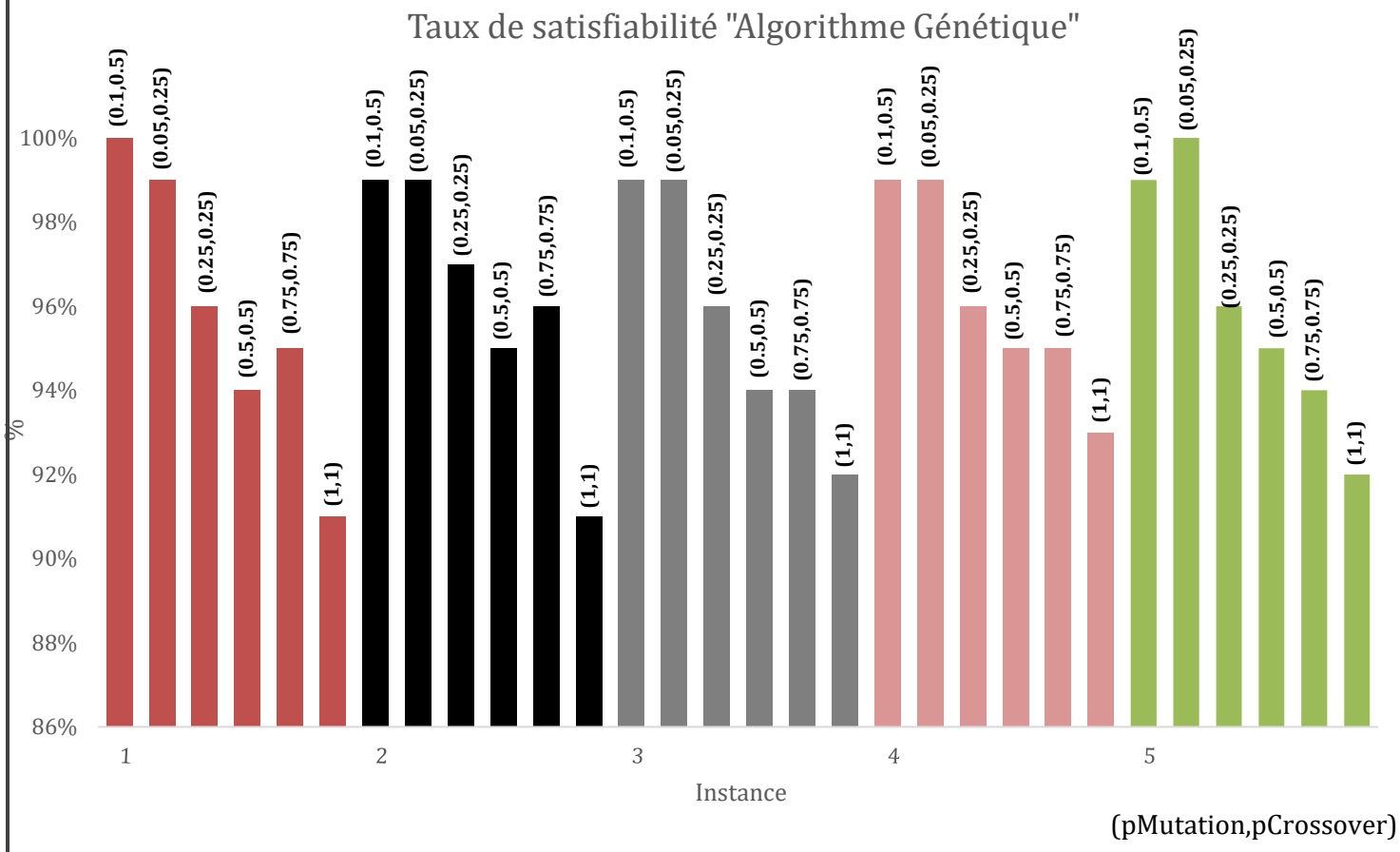
8- Complexité spatiale : $O(2^{75})$

5.2. Analyse et comparaison des résultats :

Benchmarks	Instance	Pmutation	Pcrossover	NCS	%	Temps d'exe (s)
UF75-325	1	0.1	0.5	325	100%	12,96
		0.05	0.5	323	99%	12.141
		0.25	0.25	313	96%	13.704
		0.5	0.5	307	94%	13.660
		0.75	0.75	308	95%	15.507
		1	1	296	91%	13.721
	2	0.1	0.5	323	99%	11,513
		0.05	0.5	324	99%	12.141
		0.25	0.25	314	97%	14.148
		0.5	0.5	309	95%	13.518
		0.75	0.75	312	96%	13.688

		1	1	297	91%	12.195
	3	0.1	0.5	322	99%	12,78
		0.05	0.5	323	99%	11.03
		0.25	0.25	312	96%	13.898
		0.5	0.5	307	94%	14.578
		0.75	0.75	307	94%	15.846
		1	1	299	92%	13.505
	4	0.1	0.5	321	99%	12,925
		0.05	0.5	323	99%	11.117
		0.25	0.25	313	96%	13.452
		0.5	0.5	308	95%	13.666
		0.75	0.75	308	95%	14.998
		1	1	303	93%	13.743
	5	0.1	0.5	322	99%	12.74
		0.05	0.5	325	100%	11.979
		0.25	0.25	316	96%	13.253
		0.5	0.5	309	95%	13.05
		0.75	0.75	307	94%	14.849
		1	1	299	92%	13.331
UUF75-325	1	0.1	0.5	322	99%	12.591
		0.05	0.5	323	99%	12.180
		0.25	0.25	312	96%	15.142
		0.5	0.5	307	94%	14.204
		0.75	0.75	307	94%	15.371
		1	1	298	92%	13.231
	2	0.1	0.5	322	99%	11.387
		0.05	0.5	324	99%	11.337
		0.25	0.25	313	96%	12.656
		0.5	0.5	306	94%	13.867
		0.75	0.75	306	94%	13.801
		1	1	301	93%	12.487
	3	0.1	0.5	323	99%	12.882
		0.05	0.5	322	99%	10.835
		0.25	0.25	313	96%	14.057
		0.5	0.5	306	94%	14.781
		0.75	0.75	303	93%	13.847
		1	1	298	92%	12.908
	4	0.1	0.5	320	98%	12.390
		0.05	0.5	322	99%	11.528

		0.25	0.25	312	96%	13.275
		0.5	0.5	309	95%	14.148
		0.75	0.75	308	95%	14.705
		1	1	298	92%	15.240
	5	0.1	0.5	322	99%	12.325
		0.05	0.5	322	99%	11.988
		0.25	0.25	312	96%	14.267
		0.5	0.5	308	95%	14.053
		0.75	0.75	308	95%	14.476
		1	1	301	93%	13.977



5.3. Conclusion :

Les paramètres empiriques utilisés, et qui on donner de meilleurs résultats après plusieurs expérimentations :

- Nombre de générations : 2000 générations,
- Single point crossover : Le croisement se fait au milieu,
- Population : 100,
- Nombre de parents sélectionnés : Population / 2 ie. 100 / 2 = 50.

A la fin de chaque génération la sélection de la prochaine population s'effectue en triant les parents sélectionnés, les enfants générés et la population non sélectionné. On trie tous ces derniers selon leurs valeurs de fitness et on prend les 100 premiers meilleurs triés parmi 150.

- La sélection se fait par la méthode de roulette,
- Taux de croisement : 1,
- Taux de mutation :0.05,
- Stop criteria : Génération nº 2000.

6. Partie 3 :

1- Codage de la solution :

Tableau : Représente une seule fourmi, contient la solution (sa taille = nombre de variables).

Tableau de fourmi : Contient ensemble de solutions (sa taille = nombre de fourmi).

2- Pseudo code :

```
begin
  Pheromone and parameters initialization ();
  for i=1 to Max-Iter do
      begin
        for k=1 to Nb-ants do
            begin
                Build a solution (s_k);
                Evaluate (s_k);
                Apply online delayed update of pheromone();
            end;
        Determine the best solution of the iteration();
        Apply offline delayed update of pheromone();
      end;
end.
```

```
Procedure Building (var s);
Input: L the set of all literals;
Output: s;

begin
  s = empty;
  while (L not empty) do
     select l_i from L using the transition rule;
     eliminate l_i and its opposite from L;
     Append l_i to s;
  endwhile;
end.
```

```
Procedure evaluate (s_i);
Input: s_i the solution built by ant i;
Output: f(s_i);

begin
  f_i := 0;
  C= set of all clauses;
  j :=1;
  while (j <= n) and (C not empty) do
       for each clause c satisfied by Sol[Ant_i , j] do
            f_i:=f_i + weight[c];
            eliminate c from C;
       endfor;
       j := j+1;
  endwhile;
  return f_i;
end.
```

3- La classe « Ant » :

Cette classe représente une solution de fourmi :

```
public class Ant {
    int [] solution;
    int fitness;
    boolean[] done;
    double q0=0.5;
    double alpha=0.2;
    double beta=0.6;
    double somme=0;
```

La classe contient les fonctions suivantes :

a. *contructSolution()* :

Dans notre méthode de construction, on a utilisé :

i) Dans le cas ou q<=q0 on fait appelle à la sélection par classement « argmax ».

ii) Dans le cas contraire on fait appel à la fonction de sélection roulette.

Pour faire la construction, la première position est choisie aléatoirement et ainsi que la valeur du littéral. Pour calculer les prochaines valeurs avec la probabilité, on a utilisé un vecteur « done » pour savoir si les variables sont déjà vues ou pas.

- if $q \leq q_o$ then :

$$P_{ij}^{k}(t) = \begin{cases} 1 & \text{if } j = \text{argmax}\{[T_{ij}(t)]^{\alpha}[\eta_{ij}]^{\beta}\} \\ 0 & \text{otherwise} \end{cases}$$

else

$$P_{ij}^{k}(t) = \frac{[T_{ij}(t)]^{\alpha}[\eta_{ij}]^{\beta}}{\sum_{l \in N_i^k}[T_{il}(t)]^{\alpha}[\eta_{il}]^{\beta}}$$

```
public Ant contructSolution(int [][]tab,int nbClause,int nbVar,Pheromons pheromons,double ro,double to) {
   somme=0;
    Arrays.fill(done, Boolean.FALSE);
      for (int i=0;i<nbVar;i++) {
            solution[i]=-1;
     }
   //initialiser quelle variable on va commencer avec
    Random random=new Random();
    int pos=(int) (Math.random()*nbVar);
    solution[pos]=(int) (Math.random()*2);
    done[pos] = true;
    onlineStepByStepPheromonUpdate(pos,solution[pos] , pheromons, ro,to);

     int[] argmax;
    for (int i = 0; i < nbVar-1; i++)
    {    // condition  d'arret fitness == nbVar
        if (this.calculefitnessSolution(tab, nbClause, nbVar)==nbClause)
            break;
        double q = ThreadLocalRandom.current().nextDouble();
      if (q <= q0)
        {   argmax = getArgmax(tab,nbClause,nbVar,pheromons);
                solution[argmax[0]]=argmax[1];
                 done[argmax[0]] = true;
```

```
        }else
           {
             argmax = selectionRoulette(tab, nbClause, nbVar,pheromons);
                  solution[argmax[0]]=argmax[1];
                done[argmax[0]] = true;
                }
            onlineStepByStepPheromonUpdate(argmax[0], argmax[1], pheromons, ro,to);
}
 return this; }
```

b. *selectionRoulette()* :

Nous avons utilisé la méthode de sélection par roulette, pour cela, on génère une valeur aléatoire **q**, et chaque fois on extrait le max de « argmax », qui est la probabilités la plus élevée parmi les littéraux des variables non visitées (et ceci grâce au vecteur « done » comme précisé précédemment) ; cette dernière est sommée avec la variable « partialSum », si la valeur aléatoire **q** appartient au champs de valeur de cette dernier (càd q < partialSum), ce littérale est retourné et va être affecté à la solution en cours de construction, sinon (càd q>partialSum), on extrait un autre max de « argmax » (celui qui précède le premier max), et on le somme avec « partialSum ». On refait la même chose jusqu'à ce qu'il ne reste plus de valeur a visiter ou bien si (q < partialSum), donc le dernier argmax va être retourné .

```
public int[] selectionRoulette(int [][]tab,int nbClause,int nbVar,Pheromons pheromons) {
     Ant tmp=new Ant(nbVar);
       for (int i=0;i<nbVar;i++) {
       tmp.solution[i]=this.solution[i];}
       for (int i=0;i<nbVar;i++) {
           tmp.done[i]=this.done[i];}
double totale=tmp.getProba(tab,pheromons, nbClause, nbVar);
double q = ThreadLocalRandom.current().nextDouble();
 int[] argmax=tmp.getArgmax(tab,nbClause,nbVar,pheromons);
double partialSum=tmp.getPherHeur(argmax[0], argmax[1],pheromons,tab,nbClause,nbVar)/totale;
  int compteReste=0,compte=0;
   // compter le nombre de variable qu'on doit visité
   for ( int c = 0; c < pheromons.pheromonValues.length; c++)
   {     if(done[c]==Boolean.FALSE) {
       compteReste++;
   }}
while (q > partialSum)
{ if(compte!=compteReste-1) {

     argmax =tmp.getArgmax(tab,nbClause,nbVar,pheromons);
     partialSum +=(tmp.getPherHeur(argmax[0], argmax[1],pheromons,tab,nbClause,nbVar)/totale);
      compte++;
}else return argmax;

}
return argmax;
```

c. *getArgmax()*:

C’est la fonction qui renvoie la max du produit tij^alpha*nij^beta de toute les littéraux des variables non visité.

```
 int[] getArgmax(int [][]tab,int nbClause,int nbVar,Pheromons pheromons)
{
    int[] argMax = {0, 0};
    int c=0;int k=0;

        while ((done[k]==Boolean.TRUE)&& (k< pheromons.pheromonValues.length))
        {k++;
        if (k==pheromons.pheromonValues.length-1) break;
        }
       if (done[k]==Boolean.TRUE) return argMax;

    argMax[0] = k;
    argMax[1] = 0;
    double max=getPherHeur(k, 0,pheromons,tab,nbClause,nbVar);
    double cpt ;
    for (int i = 0; i < pheromons.pheromonValues.length; i++)
    {     if(done[i]==Boolean.FALSE) {
        for (int j = 0; j <2 ; j++)
        {
            cpt=getPherHeur(i, j,pheromons,tab,nbClause,nbVar);

                if (cpt > max)
            {
                argMax[0] = i;
                argMax[1] = j;
                max=cpt;
            }
        }}
    }
```

d. *getPherHeur()* :

Calcule pour chaque littéral d’une variable, la probabilité correspondante.

$$P_{ij}^{k}(t) = \frac{[T_{ij}(t)]^{\alpha}[\eta_{ij}]^{\beta}}{\sum_{l \in N_i^k}[T_{il}(t)]^{\alpha}[\eta_{il}]^{\beta}}$$

```
double getPherHeur(int variable, int literal,Pheromons pheromons,int [][]tab,int nbClause , int nbVar) {
  Ant tmp=new Ant(nbVar);
  for (int i=0;i<nbVar;i++) {
  tmp.solution[i]=this.solution[i];}
  for (int i=0;i<nbVar;i++) {
      tmp.done[i]=this.done[i];}
  tmp.solution[variable]=literal;
  tmp.done[variable]=true;
   int mu = tmp.calculefitnessSolution(tab, nbClause, nbVar) ;
   double muBeta = Math.pow(mu, beta);
   double ti = (double)  pheromons.pheromonValues[variable][literal];
   double tiAlpha = Math.pow(ti, alpha);
   return tiAlpha * muBeta;
}
```

e. *getproba()* :

Calcule la somme de toutes les probabilités, pour faire la sélection par roulette.

```
double getProba(int [][]tab ,Pheromons pheromons,int nbClause,int nbVar)
{
     double sum = 0;
     for(int i=0;i<nbVar;i++) {
         if(done[i]==Boolean.FALSE) {
      for (int j = 0; j < 2; j++)
      {
          sum += getPherHeur(i, j,pheromons,tab,nbClause,nbVar);
      }}}
     return (double) sum;
}
```

f. *onlineStepByStepPheromonUpdate()* :

Fait la mise à jour en ligne.

```
Double onlineStepByStepPheromonUpdate(int variable, int literal,Pheromons pheromons,double ro,double to)
{

     double Ti = pheromons.pheromonValues[variable][literal];
     pheromons.pheromonValues[variable][literal] = (1 - ro) * Ti + ro * to;
     this.somme=pheromons.pheromonValues[variable][literal] -Ti;
     return pheromons.pheromonValues[variable][literal] ;
}
```

g. *calculeFitness()* :

Evalue la fourmi. La valeur de la fonction objective « *Fitness* » est égale au nombre de clauses satisfaites.

```
public int calculefitnessSolution(int [][]tab,int nbClause,int nbVar) {
    int b=0;
    int fitness=0;

    for(int j=0;j<nbClause;j++)
    {b=0;
        for(int i=0;i<nbVar;i++)
        { if(done[i]==true){
            if(((tab[j][i]==1)&&(this.solution[i]==1)) || ((tab[j][i]==0)&&(this.solution[i]==0)) )
            {fitness++; b=1;break;}
            if (b==1) break;
          }

        }

    }
    return fitness;
```

4- La classe « *Itération* » :

La classe « Itération » contient un tableau de solutions de fourmi i.e. une génération.

```
public class Iteration {

    Ant [] individu;
    int sizePopulation;
```

Dans la même classe nous avons défini deux fonctions :

a. *evalutePopulation()* :

Evalue la génération. Pour chaque fourmi on fait appel à la fonction *calculeFitness()*.

```
public void evaluatPopulation(int[][]tab, int nbClause,int nbVar) {

        for (int i=0;i<sizePopulation;i++)
        {individu[i].calculefitness(tab, nbClause, nbVar);
                    }
}
```

b. *getSBest()* :

Extraire la fourmi qui la valeur maximum de fitness.

```
public int  getSGBest(Population p) {
    int max=0;
    int pos=0;
    for (int i = 0; i < p.sizePopulation; ++i) {
        if (p.individu[i].fitness > max) {
            max = p.individu[i].fitness;
            pos=i;
        }
    }

    return pos;
}
```

5- La classe « *Phéromone* » :

Cette classe représente la phéromone.

```
public class Pheromons
{
    public Double initValue;
    public   Double[][] pheromonValues;
    public Double to;
```

Une fonction init qui initialise le tableau de phéromone avec *To.*

```
public void init(Double initValue)
{
    for (int i = 0; i < pheromonValues.length; i++)
    {
        for (int j = 0; j < pheromonValues[i].length; j++)
        {
            pheromonValues[i][j] = initValue;
        }
    }
}
```

6- La classe « *ACS*» :

La classe principale contient les méthodes suivantes :

a. *offlinePheromonUpdate()* :

La fonction qui fait la mise à jour offline, c'est la somme de toute la phéromone déposée par la formi qui a la meilleure fitness. C'est donc

*ro*to*nombre_de_variables_visitées*. La mise à jour offline est faite seulement dans les littéraux où elle est passé la meilleure fourmi.

```
public static void offlinePheromonUpdate(Ant bestAnt,Pheromons pheromons,double ro,double to)
   {       int j; int nbVarVisité=0;
  // la somme de toute la phéromone déposé donc c'est ro*to*nbr de variable visitée
  for (int i=0;i<bestAnt.solution.length;i++) {

    if(bestAnt.done[i]==true)  nbVarVisité++;
  }

       for (int i = 0; i < pheromons.pheromonValues.length; i++)
       {
           if (bestAnt.solution [i]==0) j=0;
           else j=1;

               double Ti = pheromons.pheromonValues[i][j];
               pheromons.pheromonValues[i][j]= (1 - ro) * Ti + ro*to*nbVarVisité;


       }
   }
```

7- La classe « Main » :

En premier, nous avons initialisé nos fourmis avec A*, ensuite, nous les avons évaluées et avons extrait le « *gBest* ». Nous avons également fait la mise à jour online et offline.

On rentre ensuite dans la boucle et on itère pour *nbrGeneration*.

Tout en évaluant à chaque itération le *sBest* qui est comparé au *gBest*.

- ➔ Après avoir fait plusieurs expérimentations, la sélection par roulette a pris beaucoup de temps pour construire la solution, et nos moyens matériaux n'ont pas été suffisants (bug de pc).

Et enfin, le *gBest* est retournée à la fin de l'algorithme.

```
for (int generationIndex = 0; generationIndex < nbrgeneration; generationIndex++) {
      //la contruction des nouvelles fourmis
      for (int i=0;i<nbrFourmi;i++)
      {
            Iteration.individu[i]=Iteration.individu[i].contructSolution(clauses, nbClause, nbVar, pheromons, ro, to) ;
      }
      // faire la mise a jour offline de la phéromone
      ///2 evaluate Iteration //////////////:
      Iteration.evaluatPopulation(clauses,nbClause,nbVar);
      best2=Iteration.getSGBest(Iteration);
      sbest=Iteration.individu[best2];
      offlinePheromonUpdate(sbest, pheromons, ro,to);
```

6.1. Analyse et comparaison des résultats :

Benchmarks	Instance	alpha	beta	ro	q0	NCS	%
UF75-325	1	0.2	0.6	0.6	0.5	325	100%

		0.2	**1**	**0.95**	**0.5**	**320**	**98%**
		0.3	**0.8**	**0.2057**	**0.01**	**321**	**98%**
		0.3	**0.8**	**0.2057**	**0.5**	**322**	**99%**
		0.3	**0.8**	**0.2057**	**0.8**	**319**	**98%**
		0.8	**0.2**	**0.2057**	**0.01**	**323**	**99%**
UUF75-325	**1**	**0.2**	**0.6**	**0.6**	**0.5**	**322**	**99%**
		0.2	**0.6**	**0.6**	**0.8**	**321**	**98%**
		0.3	**0.8**	**0.2057**	**0.01**	**320**	**98%**
		0.3	**0.8**	**0.2057**	**0.5**	**319**	**98%**
		0.3	**0.8**	**0.2057**	**0.8**	**311**	**95%**
		0.8	**0.2**	**0.2057**	**0.01**	**313**	**96%**

6.2. Conclusion :

Les paramètres empiriques utilisés, et qui on donner de meilleurs résultats après plusieurs expérimentations, et d'après l'état de l'art :

- Nombre d'itérations : 2000 générations,
- Nombre de fourmi : 3
- La construction se fait par la méthode de roulette, et par la méthode de classement, selon la valeur du random
- Stop criteria : Génération nº 2000.
- Valeur de alpha :0.2
- Valeur de beta :0.6
- Valeur de to :0.1
- Taux d'évaporation = 0.95
- Initialisation des fourmis avec A* : On a lancé le A*, trois fois (nombre de fourmis), et les résultats de ce dernier, on a initialisé avec nos fourmis.

```
int [] S1= {1, 1, 0, 1, 0, 1 ,0, 1, 0, 0, 0, 0, 0, 0, 0, 0 ,1, 0, 0, 0 ,0 ,0
,0 ,0 ,0 ,1 ,1 ,0 ,1, 1, 1 ,1, 0, 1,0 ,1, 0, 0, 0, 1, 0, 0, 1 ,0 ,0 ,0 ,1 ,0
,0 ,0, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0 ,1, 0, 1 ,0, 1, 1, 1,
0};
int [] S2= {1, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0,
0, 0, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0,
0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1,1, 0 ,0, 0, 0,1, 0, 1, 0 ,1 ,1, 1,
0};
int [] S3= {1 ,1 ,0 ,1, 0,1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0,
0, 0, 0, 1, 1, 0, 1 ,0 ,1 ,1 ,0 ,1, 0, 1, 0, 1, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0,
0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0,1, 0, 1,1 ,0, 0, 0, 0, 1, 0, 1, 0, 1 ,1, 1,
0};
```

```
//initialisation des fourmis avec A*
 Iteration.individu[1].solution=S1;
 Iteration.individu[2].solution=S2;
 Iteration.individu[3].solution=S3;
```

```
int [] S1= {1, 1, 0, 1, 0, 1 ,0, 1, 0, 0, 0, 0, 0, 0, 0, 0 ,1, 0, 0, 0 ,0 ,0 ,0 ,0 ,0 ,1 ,1 ,0 ,1, 1, 1 ,1, 0, 1,0 ,1, 0, 0, 0, 1, 0, 0, 1
int [] S2= {1, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 0, 1
int [] S3= {1 ,1 ,0 ,1, 0,1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1 ,0 ,1 ,1 ,0 ,1, 0, 1, 0, 1, 0, 1, 0, 0, 1,
//initialisation des fourmis avec A*
Iteration.individu[1].solution=S1;
Iteration.individu[2].solution=S2;
Iteration.individu[3].solution=S3;
```

7. Conclusion :

Nous avons traité dans ce projet le problème de satisfiabilité avec les trois types d'algorithmes. Nous avons commencé d'abord avec les algorithmes aveugles, qui n'ont pas montré de bons résultats vis-à-vis le temps d'exécution, la complexité spatiale, et nombres de clauses satisfaites, contrairement au algorithmes évolutionnaires (Algorithme Génétique) et ACS. Ces dernière dépendent de paramètres empirique, qui sont fixée en effectuant plusieurs expérimentations.