Rapport

Sujet : Résolution de sytèmes d'équations booléenes

Table des matières

Présentation du problème

Dans le cadre d'un devoir de programmation fonctionnelle à l'université, nous avons proposé une solution en langage Ocaml au problème de satisfaisabilité booléennnne.

Le problème de satisfiabilité booléenne est un problème de décision classique en informatique, qui, étant donné une formule de logique propositionnelle détermine s'il existe une instance des variables ou clauses qui vérifie la formule donné.

La problème est le suivant étant donné un système d'équations booléennes à n variables, déterminer V1,V2,…,Vn tels que le système est satisfait.

Un solveur de SAT prend souvent en argument une/des formule(s) en forme normale conjonctive (CNF).

II/ Solution algorithmique et explications

1/ Structure utilisé :

Expression booléenne :

C'est le type de base pour une équation, et donc pour un système d'équation. Il comporte une infinité de variables booléennes V_i, les constantes TRUE et FALSE, ainsi que trois connecteurs logiques à deux opérandes : OR (ou, vrai si l'une des deux opérandes est vrai), AND (et, vrai si les deux opérandes sont vraies), XOR (ou exclusif, vrai si une seule des deux opérandes est vrai). Le type possède aussi un connecteur à une opérande, NOT (sa valeur est l'inverse de la valeur de vérité de son opérande). Voici la définition de ce type en Ocaml, comme décrit dans l'énoncé du devoir :

```
type eb = V of int | TRUE | FALSE | AND of eb * eb | OR of eb * eb | XOR of eb * eb | NOT of eb;;
```

Sortie dans ocaml :

```
type eb =
    V of int
  | TRUE
  | FALSE
  | AND of eb * eb
  | OR of eb * eb
  | XOR of eb * eb
  | NOT of eb
```

Ainsi, en ocaml, pour tout n dans l'ensemble des entiers, la variable V_n s'écrit `V(n)` ; l'expression `eb1 AND eb2` s'écrit `AND (eb1, eb2)` ; l'expression `eb1 OR eb2` s'écrit `OR (eb1, eb2)` ; l'expression `eb1 XOR eb2` s'écrit `XOR (eb1, eb2)` ; l'expression `NOT eb` s'écrit `NOT (eb1)`.

Équation booléenne :

C'est un couple composé de deux expressions booléennes, simulant une égalité.

Caractérisation dans ocaml :

```
eb * eb
```

Exemples :

V_1 = V_2 : (V(1),V(2))

V_1 XOR V_2 = TRUE : (XOR(V(1),V(2)),TRUE)

Environnement (distribution de valeur de vérité):

C'est un système d'équations particulier : les équations sont toutes de la forme (V(n),TRUE/FALSE) (V(n) étant une variable, elle est donc associée à soit TRUE, soit FALSE). Chaque variable V(n) n'apparaît qu'une seule fois dans le système.

Caractérisation dans ocaml :

```
(eb * eb) list
```

Exemple :

```
[(V(1),FALSE), (V(2), TRUE), V(3),(FALSE)]
```

Système de distribution de valeur :

C'est une liste d'environnements. Tout les environnements de cette liste sont différents.

Caractérisation dans ocaml :

```
(eb * eb) list
```

Exemple :

```
[(V(1),FALSE), (V(2), TRUE), V(3),(FALSE)]
```

2/ Fonctions utiles

Les fonctions suivantes ne sont pas directement liées à l'objectif du projet mais sont utiles dans plusieurs fonctions du projet.

Fonction insertion :

fonction qui insère un couple (variable,vérité) seulement s'il n'est pas dans la liste passée en paramètre.

Cette fonction est utilisée dans la fonction « append », pour éviter d'ajouter une variable découverte dans le système d'équation en double.

La fonction parcourt récursivement toute la liste, si la variable est présente à un endroit de la liste, la liste de variable est renvoyée sans modification. Sinon la variable passée en deuxième paramètre est ajoutée à la liste, à la fin de la liste.

Code de la fonction :

```
1. let rec insertion liste var =
2. match liste with
3. |elt::ll -> if var = elt then liste else elt::(insertion ll var)
4. |[] -> var::[];;
```

La ligne 3 est la ligne clée. Si la variable var passée en second paramètre de la fonction est au début de la liste passée en premier paramètre, alors on renvoie la liste non modifiée. Sinon on cherche dans la suite de la liste. Si l'élément n'est pas dans la liste (ligne 4) on l'ajoute.

Sortie dans ocaml :

```
val insertion : 'a list -> 'a -> 'a list = <fun>
```

Jeu de test :

```
# insertion [] (V(4));;
- : eb list = [V 4]

# insertion [V(8)] (V(4));;
- : eb list = [V 8; V 4]

# insertion [V(8);V(4)] (V(4));;
- : eb list = [V 8; V 4]
```

Fonction append:

Fonction qui concatène deux listes passées en paramètre en supprimant les doublons (grâce à la fonction insertion).

C'est une fonction utilisée dans plusieurs fonctions, pour concaténer des listes pour des besoins différents. Aucunes n'ont besoin d'avoir de doublons, et certaines comme la liste générées par la fonction variable ne doivent pas posséder de doublons, c'est pourquoi la fonction les suppriment.

C'est une fonction récursive, qui ajoute tout les éléments de la première liste passée en paramètre à la deuxième liste. Pour cela, elle insert le premier élément dans la liste, puis s'appelle elle même sur le reste de la liste. Cette fonction récursive doit posséder plusieurs points d'arrêts : Si la deuxième liste est vide, alors la première liste est renvoyée. Si la première liste est vide, alors la deuxième liste est renvoyée. Si les deux listes sont vides, une liste vide est renvoyée.

Code de la fonction :

```
1. let rec append liste1 liste2 =
2. match liste1, liste2 with
3. |[],x::ll -> liste2
4. |x::ll,[] -> liste1
5. |[],[] -> []
6. |x1::ll1, x2::ll2 -> insertion (append ll1 liste2) x1;;
```

La ligne 2 correspond au match des deux listes. Les lignes 3 à 5 sont les points d'arrêts. Dans la ligne 6, la fonction insertion est appelée pour insérer le premier élément de la liste dans la concaténation du reste de liste1 et de liste2 (append ll1 liste2).

Sortie dans ocaml :

```
val append : 'a list -> 'a list -> 'a list = <fun>
```

Jeu de test :

```
# append [] [];;
- : 'a list = []

# append [8] [];;
- : int list = [8]

# append [] [8];;
- : int list = [8]
# append [1;2;3;8] [1;3;5];;
- : int list = [1; 3; 5; 8; 2]
```

1/ Recherche des variables du système

Le premier objectif est de trouver l'ensemble des variables composant le système. Les fonctions suivantes réalise ce travail.

Fonction « variable »:

Renvoie la liste des variables d'une expression booléenne.

Cette fonction est utilisée dans la fonction ens_variable. Elle permet de sous-traiter la recherche des variables d'une seule expression booléenne.

C'est une fonction récursive, qui parcourt l'arbre de l'expression booléenne passée en paramètre. Si une structure AND, OR, ou XOR est rencontrée, alors la fonction s'appelle elle même deux fois et part explorer les parties gauche et droite. Si une structure NOT est rencontrée, la fonction s'appelle une fois sur l'expression sur laquelle s'applique le NOT. La récursion s'arrête si la fonction trouve des blocs élémentaires de l'expression tels que TRUE ou FALSE ou une variable booléenne (qui est donc ajoutée à la liste).

l

```
1. let rec variable exp =
2. match exp with
3. |AND(p1,p2) -> append (variable p1) (variable p2)
4. |OR(p1,p2) -> append (variable p1) (variable p2)
5. |XOR(p1,p2) -> append (variable p1) (variable p2)
6. |NOT(p) -> variable p
7. |TRUE -> []
8. |FALSE -> []
9. |_ -> exp::[];;
```

On effectue un match sur l'équation, et on a tout les cas possible pour une expression booléenne : AND, OR, XOR, NOT, dans ce cas on continue la récursion. TRUE, FALSE, dans ce cas la fonction renvoie une liste vide (TRUE et FALSE ne peuvent pas contenir de variables booléennes). Le reste ne peut être qu'une variable booléenne, donc on renvoie une liste contenant cette variable.

Sortie dans ocaml :

```
val variable : eb -> eb list = <fun>
```

Jeu de test :

```
# variable TRUE;;
- : eb list = []

# variable (V(1));;
- : eb list = [V 1]

# variable (NOT(AND(V(1),OR(TRUE, XOR(V(2),V(1))))));;
- : eb list = [V 1; V 2]
```

Fonction « ens_variable »:

Renvoie la liste des variables utilisées dans un système d'équations booléennes.

Cette fonction permet de répondre à la question 1 du devoir. Elle renvoie la liste des variables booléennes du système.

C'est une fonction récursive simple, qui ne fait que découper toutes les équations booléennes du système d'équation, chacune en deux expressions booléennes (la partie gauche et droite de l'égalité). La liste des variables de chaque expressions est ensuite calculée (fonction « variable ») et toutes les listes sont concaténées, avec suppression des doublons (fonction « append »).

Code de la fonction :

```
1. let rec ens_variable syst =
2. match syst with
3. |(v1,v2)::ll -> append (append (variable v1) (variable v2)  (ens_variable ll)
4. |[] -> [];;
```

À la ligne 3, la fonction, pour chaque équation booléenne du système découpe cette équation en deux parties qu'elle évalue séparément avec la fonction « variable »).

Sortie dans ocaml :

```
val ens_variable : (eb * eb) list -> eb list = <fun>
```

Jeu de test :

```
#      ens_variable                [((NOT(AND(V(1),OR(TRUE,XOR(V(2),V(1)))))),(
NOT(AND(V(1),OR(TRUE,XOR(V(2),V(1)))))) ;(TRUE,FALSE);        (V(3),V(1))];;
- : eb list = [V 1; V 3; V 2]
```

Le deuxième objectif est de générer, à partir de l'ensemble des variables, tous les environnements possibles. Une fonction fait ce travail seule, la fonction environnement.

2/ Génération des environnements possibles :

Fonction « environnement »:

Fonction qui génère toutes les distributions de valeur de vérité possibles pour une liste de variables. Par exemple, prenons une liste arbitraire de 3 variables booléennes : [V_1 ; V_2 ; V_3]. Alors la fonction doit générer une liste de $2^3 = 8$ distributions de valeurs de vérité possibles, par exemple : (V_1=FALSE ; V_2=FALSE ; V_3=FALSE), (V_1=FALSE ; V_2=FALSE ; V_3=TRUE), (V_1=FALSE ; V_2=TRUE ; V_3=FALSE), etc.

Cette fonction permet de répondre à la question 2 du devoir, en renvoyant la liste des distributions de valeurs de vérité.

On créé une fonction récursive qui possède deux arguments : une liste de variables et un accumulateur. Au départ, l'accumulateur est une liste vide, la liste de variables contient toutes les variables booléennes du système d'équation. La fonction créée, à partir de l'accumulateur qu'elle reçoit en paramètre (qu'on appellera A), et de la première variable de la liste de variables (qu'on appellera V), deux accumulateurs : un qui contient A, auquel on ajoute le couple (V, TRUE) (donc V qui est vrai dans cet accumulateur), et un autre accumulateur qui contient A auquel on ajoute le couple (V, FALSE). Si V était la dernière variable du système, on renvoie les deux accumulateurs, qui constituent une distribution de valeur car toutes les variables du système sont dedans. Sinon, on appelle la fonction avec les deux accumulateurs créés, et avec la liste de variables privée de V. On a bien toutes les distributions de valeurs possibles, car la fonction créée, à chaque niveau de profondeur, 2 fois plus de distributions de valeurs, donc pour n variable(s) elle créée bien 2^n distributions de valeur différentes.

Code de la fonction :

```
1. let rec environnement var a =
2. match var with
3. |v::[] -> ((v,true)::a)::(((v,false)::a)::[])
4. |v::ll -> append ((environnement ll ((v,true)::a))) ((environnement ll      ((v,false)::a)));;
```

La ligne 3 marque le point d'arrêt de la récursion, lorsqu'il n'y a plus qu'une variable dans la liste, on créé une « liste de listes de couples » (liste de distribution de valeur), qui possède deux distributions. Sinon on continue la récursion, en appelant deux fois la fonction. Il ne faut pas oublier de passer en deuxième paramètre un accumulateur vide initialement.

Sortie dans ocaml :

```
val environnement : 'a list -> ('a * bool) list -> ('a * bool) list list = <fun>
```

Jeu de test :

```
# environnement [V(1)] [];;
- : (eb * bool) list list = [[(V 1, true)]; [(V 1, false)]]

# environnement [V(1);V(8);V(2)] [];;
- : (eb * bool) list list =
[[(V 2, true); (V 8, false); (V 1, false)];
[(V 2, false); (V 8, false); (V 1, false)];
[(V 2, false); (V 8, true); (V 1, false)];
[(V 2, true); (V 8, true); (V 1, false)];
 [(V 2, true); (V 8, true); (V 1, true)];
[(V 2, false); (V 8, true); (V 1, true)];
 [(V 2, false); (V 8, false); (V 1, true)];
 [(V 2, true); (V 8, false); (V 1, true)]]
```

Après avoir trouvé la liste des variables, la liste des solutions potentielles, il faut trouver les solutions du système.

3/ Recherche des solutions du système :

Fonction « recherche_var »:

Renvoie la valeur d'une variable dans une distribution de valeur de vérité donnée.

Cette fonction est utilisée dans la fonction eval_exp, qui évalue une expression dans une distribution de valeur de vérité.

C'est une fonction récursive simple, qui prend en paramètre une liste de couples dont le premier élément est une variable du système d'équations, et le deuxième est un booléen (donc un environnement). L'autre paramètre de la fonction est la variable du système d'équations à évaluer, la fonction parcourt récursivement la liste et renvoie la valeur de vérité correspondante dès qu'elle est trouvée. La fonction s'arrête forcement car la variable passée en second paramètre doit être une variable du système d'équations de départ et l'environnement possède toutes les variables du système d'équations.

Code de la fonction :

```
1. let rec recherche_var dist equ =
2. match dist with
3.  |(var,ver)::ll -> if var = equ then ver else recherche_var ll equ;;
```

La fonction parcourt la liste à la recherche de la variable, dès que celle ci est trouvée elle est renvoyée.

Sortie dans ocaml :

```
val recherche_var : ('a * 'b) list -> 'a -> 'b = <fun>
```

Jeu de test :

```
# recherche_var [(V(5),TRUE)] (V(5));;
- : eb = TRUE

# recherche_var [(V(5),TRUE);(V(9),FALSE)] (V(9));;
- : eb = FALSE
```

Fonction « eval_exp »:

Fonction qui évalue une expression booléenne dans un environnement. La fonction prend donc deux paramètres.

C'est une fonction récursive qui va évaluer si une expression est vrai ou fausse. Pour cela, elle va remplacer toutes les variables booléennes par leur valeur selon la distribution de valeur passée en paramètre. Ensuite, elle va évaluer récursivement chaque structure de l'expression (par exemple, `P1 AND P2` est vrai seulement si `P1` et `P2` sont vrai, donc on évalue `P1` et `P2`)

Code de la fonction :

```
1. let rec eval_exp exp dist =
2. match exp with
3. |AND(p1,p2) -> (eval_exp p1 dist) && (eval_exp p2 dist)
4. |OR(p1,p2) -> (eval_exp p1 dist) || (eval_exp p2 dist)
5. |XOR(p1,p2) -> ((eval_exp p1 dist) || (eval_exp p2 dist)) && not
((eval_exp p1 dist) && (eval_exp p2 dist))
6. |NOT(p) -> not (eval_exp p dist)
7. |TRUE -> true
8. |FALSE -> false
9. |_ -> recherche_var dist exp;;
```

Les lignes de 3 à 8 ne font que transformer des types « eb » en types booléens. La ligne 9 indique que si l'expression est une variable (seul élément non testé dans les conditions du dessus) alors la fonction transforme cette variable en « true » ou « false » (de type booléen) selon l'environnement « dist » (en paramètre).

Sortie dans ocaml :

```
val eval_exp : eb -> (eb * bool) list -> bool = <fun>
```

Jeu de test :

On teste XOR car c'est la structure la plus complexe :

```
# eval_exp  (XOR(V(1),V(2))) [(V(1), true); (V(2), true)];;
- : bool = false

# eval_exp  (XOR(V(1),V(2))) [(V(1), true); (V(2), false)];;
- : bool = true

# eval_exp  (XOR(V(1),V(2))) [(V(1), false); (V(2), false)];;
- : bool = false
```

Fonction « eval_sys_equ »:

Cette fonction prend un système d'équations en paramètre, qu'elle évalue dans un environnement passé en deuxième paramètre.

Cette fonction va évaluer les équations du système d'équations dans un environnement, c'est à dire qu'elle va évaluer la valeur de vérité du membre de gauche de l'équation et celle du membre de droite. Si les deux membres ont même valeur de vérité, alors l'équation est vrai, et on teste l'équation suivante. Si toutes les équations du système sont vrai, alors la fonction renvoie « true », ce qui veut dire que l'environnement passé en paramètre est une solution du système d'équations.

Code de la fonction :

```
1. let rec eval_sys_equ sys_equ dist =
2. match sys_equ with
3. |(egaliteGauche, egaliteDroite)::ll -> if (eval_exp egaliteGauche dist)                    =
(eval_exp egaliteDroite dist) then eval_sys_equ ll dist else false 4. |[] -> true;;
```

La fonction est récursive afin de parcourir toutes les équations du système. À la ligne 3, l'équation (sous la forme (exp1, exp2)) est découpée en deux expressions qui sont évaluées séparément grâce à la fonction eval_exp. Si les deux expressions sont égales (les deux valent false, ou les deux valent true), alors on teste sur l'équation suivante. Sinon, la fonction s'arrête ici et renvoie false. La ligne 4 indique que s'il ne reste plus d'équation à tester (donc si toutes les équations sont vrais), alors cette distribution de valeur de vérité est une solution du système.

Sortie dans ocaml :

```
val eval_sys_equ : (eb * eb) list -> (eb * bool) list -> bool = <fun>
```

Jeu de test :

```
# eval_sys_equ [(OR(V(1),V(2)),TRUE); (XOR(V(1),V(3)),V(2));
(NOT(AND(AND(V(1),V(2)),V(3))),TRUE)] [(V(1),false);(V(2),false);
(V(3),false)];;
- : bool = false
```

(on vérifie aisément que c'est effectivement faux)

```
# eval_sys_equ [(OR(V(1),V(2)),TRUE); (XOR(V(1),V(3)),V(2));
(NOT(AND(AND(V(1),V(2)),V(3))),TRUE)] [(V(1),false);(V(2),true);(V(3),true)];;
- : bool = true
```

Fonction « dist_correct »:

Cette fonction prend un système d'équations en paramètre, un système de distributions de valeurs en second paramètre et évalue quelles distributions de valeurs sont solutions du système d'équations.

La fonction évalue le système d'équations dans chaque environnement, et renvoie une liste de tous les environnements solutions du système.

Code de la fonction :

```
1. let rec dist_correct sys_equ sys_dist =
2. match sys_dist with
3. |di::ll -> if ((eval_sys_equ sys_equ di) = true) then di::(dist_correct sys_equ ll)
else dist_correct sys_equ ll
4. |[] -> [];;
```

La fonction parcourt récursivement la liste sys_dist, qui contient tous les environnements, et évalue le système avec chaque environnement grâce à la fonction « eval_sys_equ ». Si cette fonction renvoie true, alors la distribution de valeur est une solution du système d'équations, elle est alors ajoutée à la liste.

Sortie dans ocaml :

```
val dist_correct : (eb * eb) list -> (eb * bool) list list -> (eb *  bool) list list = <fun>
```

Jeu de test :

```
# dist_correct [(V(1),TRUE)] [[(V(1),true)]; [(V(1),false)]];;
- : (eb * bool) list list = [[(V 1, true)]]
```

Un autre test est fait avec la fonction ci-dessous.

Fonction « solveur »:

Cette fonction est la fonction qui assemble toutes les autres, et qui répond au devoir. Pour un système d'équations donné, la fonction renvoie une liste d'environnements (distribution de valeur de vérité).

La fonction utilise les fonctions créés précédemment pour trouver l'ensemble des variables du système, générer toutes les distributions de valeurs possibles pour ces variables et enfin trouver les environnements solutions, afin de faciliter le travail de l'utilisateur.

Code de la fonction :

```
1. let solveur sys_equ =
2. dist_correct sys_equ (environnement (ens_variable sys_equ) []);;
```

Le code s'explique de lui même, les différentes fonctions sont imbriquées entre elles et chaque fonction utilise celle du niveau inférieur. À noter que la liste vide est utilisée pour la fonction environnement, qui a besoin d'un accumulateur vide pour démarrer.

Sortie dans ocaml :

```
val solveur : (eb * eb) list -> (eb * bool) list list = <fun>
```

Jeu de test :

Pour le système :

```
| V₁ OR V₂ = TRUE
| V₁ XOR V₃ = V₂
| NOT (V₁ AND (V₂ AND V₃ )) = TRUE
```

On effectue le test suivant :

```
        # solveur [(OR(V(1),V(2)),TRUE); (XOR(V(1),V(3)),V(2));
(NOT(AND(AND(V(1),V(2)),V(3))),TRUE)];;
        - : (eb * bool) list list =
        [[(V 2, true); (V 1, true); (V 3, false)];
         [(V 2, false); (V 1, true); (V 3, true)];
         [(V 2, true); (V 1, false); (V 3, true)]]
```

Il y a 3 solutions selon le programme :

```
1) [V1 = TRUE, V2 = TRUE, V3 = FALSE]
2) [V1 = TRUE, V2 = FALSE, V3 = TRUE]
3) [V1 = FALSE, V2 = TRUE, V3 = TRUE]
```

Vérification des solutions :

1)

```
      | TRUE OR TRUE = TRUE   [OK]
      | TRUE XOR FALSE = TRUE   [OK]
      | NOT (TRUE AND (TRUE AND FALSE )) = TRUE ⇒ NOT (FALSE) = TRUE   [OK]
```

2)

```
      | TRUE OR FALSE = TRUE   [OK]
      | TRUE XOR TRUE = FALSE   [OK]
      | NOT (TRUE AND (FALSE AND TRUE )) = TRUE ⇒ NOT (FALSE) = TRUE   [OK]
```

3)

```
      | FALSE OR TRUE = TRUE   [OK]
      | FALSE XOR TRUE = TRUE   [OK]
      | NOT (FALSE AND (TRUE AND TRUE )) = TRUE ⇒ NOT (FALSE) = TRUE   [OK]
```

Sur ce test le programme fonctionne correctement (le code au complet est disponible en fin d'annexe).

Conclusion & Discussion

1/ Complexité de l'algorithme utilisé :

- Soit n le nombre de variables dans notre système d'équations alors la fonction qui récupère les variables a une complexité en temps O(n).
- La génération de tous les environnements possibles avec deux valeurs possibles c'est-à-dire Vrai ou Faux prend un temps O(2^n).
- La comparaison entre la distribution de table de vérité générée et le système d'équations se fait donc en temps exponentiel.
- La complexité totale est donc en O(2^n).

Avec une étude de complexité plus détaillée du problème on peut prouver que ce problème (n-SAT) est NP pour n>3 c'est-à-dire pour un sytème d'équations à plus de trois litteraux par close.

Le problème 2-SAT, c'est-à-dire avec deux litteraux par close, est par contre lui dans la classe P, donc il peut etre résolu en temps polynomial. Des algorithmes s'appuyant sur la théorie des graphes existent pour résoudre ce genre de problème.

2/ Améliorations possibles :

- Sauter les branches où des solutions ne peuvent pas être trouvées (voir figure dans annexes).
- Ordonner la recherche pour maximiser l'espace de recherche qu'on peut sauter.

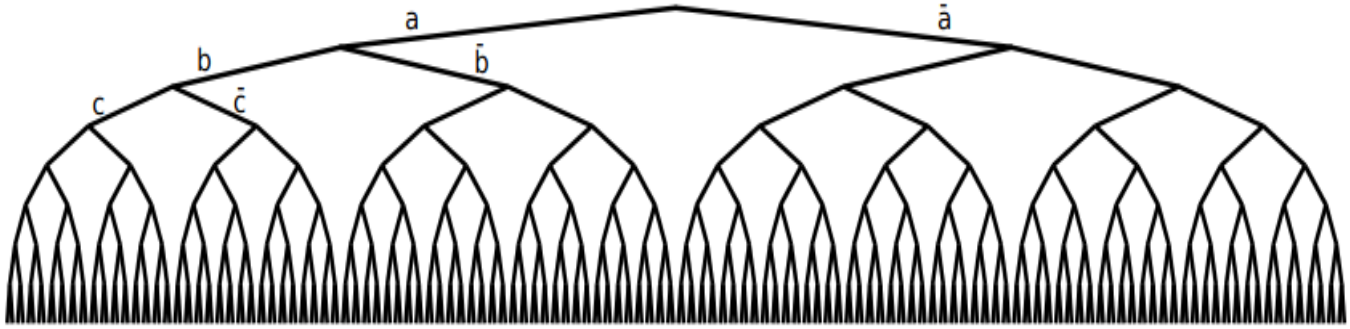
3/ Alternatives algorithmiques plus réalistes :

Un algorithme qui est souvent utilisé pour le problème des SAT est l'algorithme DPLL.
Un algorithme concu au début du 20ème siecle. Cet algorithme est plus efficient et surtout plus rapide. Cet algorithme est utilisé dans tous les systèmes modernes lors de la résolution de problèmes de planification classique, *model checking*, ou encore diagnostic.

Un résumé de son fonctionnement est disponible en annexe.

Annexe

- Arbre des solutions possibles



- Algorithme DPLL (**Davis–Putnam–Logemann–Loveland)**

L'algorithme prend en entrée une formule de la logique propositionnelle en forme normale conjonctive. L'algorithme est basé sur une méthode de *backtracking*. Il procède en choisissant un littéral, lui affecte une valeur de vérité, simplifie la formule en conséquence, puis vérifie récursivement si la formule simplifiée est satisfaisable. Si c'est le cas, la formule originale l'est aussi, dans le cas contraire, la même vérification est faite en affectant la valeur de vérité contraire au littéral. Dans la terminologie de la littérature DPLL, c'est la conséquence d'une règle dite *splitting rule (règle de séparation)*, et sépare le problème en deux sous problèmes.

L'étape de simplification consiste essentiellement en la suppression de toutes les clauses rendues *vraies* par l'affectation courante, et tous les littéraux déduits *faux* à partir de l'ensemble des clauses restantes.

L'algorithme DPLL se base sur deux principes importants :

- La propagation unitaire : Qui consiste en la suppression d'exploration de quelques branches où il ne peut pas y avoir de solutions. En théorie , une clause unitaire est une clause qui ne contient qu'un seul litéral donc elle ne peut être vrai qu'en lui affectant la valeur qui la rend vraie.

- L'élimination des littéraux purs : Si une variable propositionnelle apparaît seulement sous forme positive ou seulement sous forme négative alors ses littéraux sont dits *purs*. Les littéraux purs peuvent être affectés d'une manière qui rend toutes les clauses qui les contiennent *vrais*. Par conséquent ces clauses ne contraignent plus l'espace de recherche et peuvent être éliminées.