Algorithmes de calcul de résolvant

Laurent Simon

Rapport de Stage
Sous la direction de Philippe Chatalic

DEA Informatique,

Laboratoire de Recherche en Informatique,
Université Paris-Sud, 91405, Orsay Cedex

Lundi 8 Septembre 1997

Table des matières

Chapitre 1

Introduction

Toute modélisation formelle d'un système physique et de son comportement pose le problème de la validation du modèle construit. Dans un système complexe, certaines informations sont plus accessibles que d'autres à l'expert. Il peut s'agir de paramètres décrivant des données issues de capteurs, de paramètres d'entrée ou de sortie du système. Une approche possible de la phase de validation consiste à faire abstraction de la valeur des paramètres internes du système, pour se concentrer sur celle des paramètres les plus accessibles. Il est ainsi possible de tester si, pour certaines configurations d'entrée, le modèle produit bien en sortie les valeurs attendues par l'expert. On peut également utiliser cette technique pour rechercher des contraintes sur certains paramètres d'entrée afin d'obtenir des valeurs précises pour certains paramètres de sortie.

Lorsque le système physique est modélisé à l'aide d'une formule propositionnelle, cette méthode consiste à calculer le *résolvant* de la formule modélisant le système, par rapport à un sous-ensemble des variables propositionnelles utilisées pour décrire le système complet. Cette technique revient en quelque sorte à effectuer une projection de la formule décrivant le système, sur l'espace des variables que l'on souhaite plus particulièrement étudier. Une telle opération constitue naturellement une perte d'information mais le résolvant ainsi obtenu reste cependant une conséquence logique de la formule initiale. La confrontation du résolvant obtenu avec la réalité observable nécessite, en cas de désaccord, de remettre en cause, soit le modèle du système, soit la validité des capteurs.

Le travail présenté dans ce rapport trouve son origine dans le cadre d'une collaboration de recherche avec Électricité de France, dont le but initial est de mettre en œuvre un algorithme de calcul de résolvant et de l'appliquer à des problèmes concrets de validation de systèmes physiques. Au delà de ces exemples, il nous a semblé souhaitable d'élargir le champ d'expérimentation de l'algorithme, afin d'en mieux cerner les points forts et les limites. Pour cela, nous avons tenté de l'appliquer tant sur des exemples générés aléatoirement que sur des exemples utilisés comme référence dans la littérature concernant la résolution du problème de la satisfiabilité d'une formule propositionnelle (SAT).
En effet, l'algorithme de calcul de résolvant peut être vu comme une variante de la procédure de Davis et Putnam, que l'on retrouve au cœur de toutes les meilleures méthodes systématiques actuelles de résolution du problème SAT. Or, durant ces cinq dernières années, le traitement du problème SAT a connu un net regain d'intérêt et une amélioration significative des exemples traitables, autant par des méthodes systématiques que par des méthodes stochastiques. Pourtant, mis à part les récents travaux de R. Dechter et I. Rish ([9]), rien n'a encore été fait pour exploiter et surtout mieux comprendre les particularités de l'algorithme que nous présentons.
Ceci est d'autant plus étonnant que l'algorithme initial de démonstration automatique de théorèmes présenté en 1960 par M. Davis et H. Putnam ([7]) était en fait un cas particulier du calcul de résolvant. Une raison de cette mise à l'écart semble être due à la présentation en 1962 par M. Davis, G. Logeman et D. Loveland ([18]) d'une variante de l'algorithme initial donnant de meilleurs résultats. C'est en fait cette variante qui a retenu l'attention et que l'on nomme encore aujourd'hui, par abus de langage, *algorithme de Davis et Putnam*. L'algorithme originel a quant à lui été oublié, alors que les auteurs précisaient,

toujours dans [18], que chacune des versions avaient des caractéristiques désirables.

Ce rapport a pour but, d'une part, de situer le problème de calcul de résolvant dans la littérature actuelle, d'autre part, d'en présenter une implémentation suffisament efficace pour permettre une sérieuse étude expérimentale. Il montre par ailleurs comment cet algorithme peut être utilisé dans un vaste champ d'applications, allant de la compilation de bases de connaissances à la résolution du problème SAT. Enfin, il présente plusieurs directions qui nous semblent prometteuses pour améliorer le calcul de résolvant.

Le plan de ce rapport est le suivant : dans le second chapitre, nous définissons formellement ce qu'est un résolvant d'une formule propositionnelle, nous présentons un algorithme permettant de l'obtenir et situons cet algorithme par rapport à d'autres algorithmes proches de la littérature. Le troisième chapitre décrit la mise en oeuvre de l'algorithme, souligne l'impact des heuristiques et des structures de données utilisées sur les performances obtenues et rend compte d'une première expérimentation, tant sur des exemples réels (issus de modélisations de systèmes physiques) que sur des exemples aléatoires. Dans le quatrième chapitre, nous appliquons l'algorithme à divers autres problèmes, sur lesquels il peut également s'avérer utile. Nous montrons notamment que le calcul de résolvant, bien qu'en apparence beaucoup plus complexe que l'algorithme dit de *Davis et Putnam*, peut également être utilisé pour résoudre des problèmes de satifiabilité, et qu'il dépasse sur certaines classes d'exemples, parfois d'un facteur temps significatif, les performances des meilleurs algorithmes systématiques actuels. Ensuite, le chapitre cinq présente une approche hybride de l'algorithme, couplant les deux versions de l'algorithme de Davis et Putnam. Le chapitre six propose un travail plus prospectif, dans lequel nous étudions les particularités d'algorithmes plus spécifiques, utilisés dans le domaine des études de fiabilité, et qui sur certains exemples dépassent largement les performances du calcul de résolvant. Nous proposons une adaptation des techniques employées dans ces systèmes pour définir plusieurs nouveaux algorithmes de calcul de résolvant. Enfin, nous suggérons diverses voies de recherche qui nous paraissent prometteuses pour la suite du travail présenté.

Chapitre 2

Le calcul de résolvant

2.1 Le résolvant en calcul propositionnel

Dans ce chapitre, nous définissons formellement ce qu'est le résolvant d'une formule, dans le cadre de la logique propositionnelle. Après un rappel sur les fondements de la logique propositionnelle, et sur les principes du calcul de résolvant que nous illustrons sur un exemple simple issu de la modélisation d'un système physique, nous établissons un lien avec DPL, la principale procédure utilisée dans la littérature pour prouver la satisfiabilité d'une formule propositionnelle. Puis, nous tentons de situer le problème présenté ici par rapport à d'autres problèmes, beaucoup plus classiques dans la littérature, pour finir sur une présentation des récents travaux de I. Rish et R. Dechter, qui tendent aussi à réintroduire la première version de l'algorithme de Davis et Putnam.

Dans tout le rapport, afin de dénoter sans ambiguïté les deux versions de l'algorithme, nous appelons DP l'algorithme de calcul de résolvant présenté en 1960 dans [7] et DPL l'algorithme modifié, présenté en 1962 dans [18].

2.1.1 La logique propositionnelle

Le travail étudié dans ce rapport rentre dans le cadre des travaux de la logique propositionnelle classique. Nous en donnons ici un survol rapide, mais suffisamment complet pour une bonne compréhension du rapport. Des descriptions exhaustives de la logique des propositions sont légions dans tous les ouvrages informatiques. On peut cependant, par exemple, consulter [15, 4] pour une description complète de la logique présentée ici.

- Une *variable propositionnelle* est représentée par un symbole (généralement une lettre minuscule). Quand il n'y a pas d'ambiguité, on parle seulement de variable pour désigner une variable propositionnelle.
- Un *littéral* est une variable propositionnelle ou sa négation (par exemple p ou $\neg p$).
- Une *formule propositionnelle* est bâtie par induction d'après les règles usuelles :
  - Les constantes 1 et 0 sont des formules propositionnelles.
  - Une variable est une formule propositionnelle.
  - Si f et g sont des formules propositionnelles, alors
    - $\neg f$ (négation),
    - $f \vee g$ (ou logique),
    - $f \wedge g$ (et logique),
    - $f \rightarrow g$ (implication logique) et
    - $f \leftrightarrow g$ (équivalence logique)

sont des formules propositionnelles. De même que pour les variables, on appellera simplement formule une formule propositionnelle quand il n'y a pas d'ambiguité.

- Une *clause* est la disjonction des littéraux la composant (par exemple $(p \vee \neg q \vee r)$). Une clause peut être représentée comme l'ensemble de ses littéraux.
- Une formule sous *forme normale conjonctive* (FNC) f est une conjonction de clauses ($(a \vee b) \wedge (\neg b \vee c)$ est sous FNC).
- Une formule sous *forme normale disjonctive* (FND) f est une disjonction de conjonctions de littéraux. Une conjonction de littéraux peut alors être appelé *produit* de littéraux ($(a \wedge b) \vee (\neg b \wedge c)$ est sous FNC).
- Une clause C *subsume* une clause C' si les littéraux de C sont un sous-ensemble des littéraux de C' (par exemple, $(a \vee b)$ subsume $(a \vee b \vee c)$). On peut remarquer qu'alors $C \rightarrow C'$.

On introduit ensuite sur la syntaxe définie ci dessus la sémantique traditionnelle du calcul propositionnel :

- Une variable propositionnelle ne peut prendre que les valeurs 1 et 0 (parfois on préfère dire *vrai* et *faux*).
- Une *interprétation* d'une formule propositionnelle f est une fonction associant à chaque variable de f une unique valeur booléenne (1 ou 0) et à chaque opérateur logique défini ci-dessus une unique fonction booléenne. L'interprétation d'une formule logique est soit 0, soit 1. Dans le cadre de ce rapport, les connecteurs logiques sont interprétés par leurs fonctions booléennes usuellement associées.
- Un *modèle* d'une formule booléenne est une affectation des variables la composant dont l'interprétation est égale à 1.
- Une formule est *satisfiable* si au moins une de ses interprétations est égale à 1 (par exemple, $(a \vee b) \wedge (\neg a \vee b)$ est satisfiable pour l'affectation $(a, 1), (b, 1)$).
- Une formule est une *tautologie* si toutes ses interprétations sont égales à 1.
- Deux formules sont *équivalentes* si elles ont exactement les mêmes modèles.
- On note $f_{x \leftarrow 1}$ (resp. $f_{x \leftarrow 0}$) la fonction f dans laquelle la variable x est interprétée à 1 (resp. 0). Une telle opération est appelée *restriction* de la formule f.

Il reste à définir le calcul de résolvant sur ces définitions de base de la logique propositionnelle.

2.2 Résolvant d'une formule propositionnelle

Étant donnée une formule f construite sur un ensemble de variables V, le résolvant de f par rapport à un sous ensemble U de V peut être vu comme une formule exprimant toutes les contraintes liant les variables de U, indépendamment de la valeur des variables de $V \setminus U$. Dans le cas où $V = U \cup \{x\}$, on cherche à exprimer ce que l'on peut déduire de f indépendamment de x. Ceci peut être fait en remarquant que dans toute interprétation I satisfaisant f, la variable x est interprétée soit à 1, soit à 0. On peut donc en déduire que soit $f_{x \leftarrow 1}$ est satisfaite par I, soit $f_{x \leftarrow 0}$ est satisfaite par I, donc $f_{x \leftarrow 1} \vee f_{x \leftarrow 0}$ est satisfaite par I. Cette dernière formule exprime bien les contraintes sur les variables de U découlant des contraintes exprimées par f sur les variables de V. Le calcul de résolvant sur un ensemble de variables U consiste à appliquer récursivement la formule que nous venons de définir sur f, sur toutes les variables de $V \setminus U$. Ce résolvant est donc bien une formule définie uniquement sur les variables de U.

Pour formaliser davantage le calcul de résolvant, rappelonsons ce qu'on appelle *l'expansion de Shannon* d'une formule f par rapport à la variable x. Cette identité s'écrit

$$f = (x \wedge f_{x \leftarrow 1}) \vee (\neg x \wedge f_{x \leftarrow 0})$$

Dans le cas où la variable x fait partie des variables de f à éliminer, on peut utiliser l'expansion de Shannon ainsi définie pour éliminer x. Ainsi, si x est interprété à 1, l'expansion de Shannon permet de déduire $f_{x\leftarrow 1}$ de f. Si x est interprétée à 0, alors f s'écrit $f_{x\leftarrow 1}$. On retrouve le résultat pressenti ci-dessus, c'est-à-dire que l'élimination de x dans f s'écrit :

$$Elim_x(f) = f_{x\leftarrow 1} \vee f_{x\leftarrow 0}$$

2.2.1 Premier algorithme

Le calcul de résolvant peut se définir par des appels récursifs de $Elim_{x_k}$ sur toutes les variables x_k à éliminer. On peut donc écrire un premier algorithme comme suit, où F est une formule propositionnelle et V est l'ensemble des variables à éliminer par calcul de résolvant.

Fonction **Resolvant_Shannon(V,f)** *:*

- *Si* $V = \emptyset$*, renvoyer* f
- *Sinon, choisir une variable* v *dans* V*,*
- *renvoyer* $(Resolvant\_Shannon(V \setminus \{v\},\ f_{v\leftarrow 1}\ )\ \vee\ Resolvant\_Shannon(V \setminus \{v\},\ f_{v\leftarrow 0}\ ))$

En fait, l'algorithme initial de Davis et Putnam, proposé en [7], correspond à la fonction *Resolvant_Shannon* appliquée à l'ensemble complet des variables propositionnelles définissant la formule. Dans ce cas, la fonction renvoie une formule dont toutes les variables ont été éliminées, ce qui correspond à une formule égale à 0 (la formule est insatisfiable) ou 1 (la formule est satisfiable).

2.2.2 Résolvant sur des formes normales conjonctives

L'algorithme que nous venons de présenter peut s'appliquer directement à toute formule propositionnelle. Or, dans le cadre de ce rapport, on se restreint aux formules mises sous forme normale conjonctive. Cette restriction sur la forme des formules en entrée de la fonction permet d'exprimer le calcul de résolvant de manière plus précise sur des ensembles de clauses, en sachant que théoriquement (ce calcul peut s'avérer très coûteux), toute formule peut être réécrite en une formule équivalente, mise sous forme normale conjonctive.

Soit f une formule propositionnelle mise sous FNC. Soit C^+ (resp. C^-) la conjonction des clauses de f contenant x (resp. $\neg x$). Soit C' la conjonction des clauses de f ne contenant ni x ni $\neg x$. Alors, la formule f s'écrit en fait :

$$f = C^+ \vee C^- \vee C'$$

En passant par l'expansion de Shannon sur la variable x, on peut écrire ;

$$f = (\ (C^+_{x\leftarrow 1} \wedge C^-_{x\leftarrow 1} \wedge C'_{x\leftarrow 1}) \wedge x) \vee (\ (C^+_{x\leftarrow 0} \wedge C^-_{x\leftarrow 0} \wedge C'_{x\leftarrow 0}) \wedge \neg x)$$

ou, en simplifiant les clauses vérifiées par les affectations de la variable x (élimination des clauses satisfaites par l'affectation) :

$$f = (\ (C^-_{x\leftarrow 1} \wedge C') \wedge x) \vee (\ (C^+_{x\leftarrow 0} \wedge C') \wedge \neg x)$$

Il reste à éliminer la variable x dans la formule obtenue comme dans la section précédente et à factoriser C' pour obtenir le résolvant :

$$Resolvant_x(f) = C' \wedge (C^-_{x\leftarrow 1} \vee C^+_{x\leftarrow 0})$$

Dans la suite du rapport, nous appelons *distribution* l'opération de disjonction sur $C^-_{x\leftarrow 1}$ et $C^+_{x\leftarrow 0}$. Cette opération s'effectue en utilisant les propriétés de distributivité des connecteurs $\vee$ et $\wedge$ afin d'obtenir une forme normale conjonctive d'après la disjonction de deux formules mises sous forme normale conjonctive.

On peut remarquer que dans le cas du calcul sur des FNC, l'opération de distribution peut être retrouvée en effectuant toutes les *résolutions* sur la variable à éliminer. Nous définissons maintenant quelques termes classiques utilisés en calcul propositionnel pour trouver une autre manière d'exprimer le calcul de résolvant sur une formule mise sous FNC.

On appelle *résolution sur la variable* x entre deux clauses $(x \vee A)$ et $(\neg x \vee B)$ (où A et B sont des disjonctions de littéraux) l'opération donnant la clause $(A \vee B)$.

On appelle *coupure sur la variable* x l'opération consistant à faire toutes les résolutions sur toutes les clauses contenant la variable x.

On appelle *résolvant* par rapport à une variable x l'union des clauses ne contenant ni x ni $\neg x$ avec les clauses produites par la coupure sur x. On peut aussi trouver une autre définition du résolvant, dans laquelle on ajoute aux clauses produites les clauses qui contiennent la variable x. Dans ce cas, la variable x n'est pas éliminée.
Notons que si la variable x n'apparaît que positivement ou négativement, alors l'ensemble des clauses produites est vide et l'opération de calcul de résolvant est réduite à l'élimination des clauses contenant la variable x.

On appelle *résolvant* par rapport à un ensemble de variable V, la formule résultant de l'application récursive du calcul de résolvant sur toutes les variables de V.

Par exemple, le résolvant par rapport à la variable b de la formule

$$(a \vee b \vee c) \wedge (b \vee \neg d) \wedge (b \vee c) \wedge (\neg b \vee c \vee d) \wedge (\neg a \vee c)$$

est la formule

$$(\neg a \vee c) \wedge (a \vee c \vee d) \wedge (c \vee d) \wedge (c \vee \neg d \vee d)$$

Il est possible déliminer la clause $(a \vee c \vee d)$ car elle est subsumée par la clause $(c \vee d)$. Nous pouvons aussi enlever du résolvant la clause $(c \vee \neg d \vee d)$ qui est une tautologie. On obtient finalement la formule suivante :

$$(\neg a \vee c) \wedge (c \vee d)$$

Pour obtenir un calcul de résolvant efficace, on voit tout de suite l'importance des tests de subsomption et de tautologie. Il ne faut donc pas se contenter de calculer le résolvant, il faut aussi manipuler efficacement les ensembles de clauses produites.

2.2.3 Algorithme de calcul de résolvant

Lors de la formalisation du calcul de résolvant sur des formules mises sous FNC, le point clé de l'algorithme est dans la distribution des clauses. On peut donner l'algorithme mettant à jour cette distribution comme une fonction récursive prenant comme argument un ensemble F de clauses et un ensemble V de variables à éliminer.

Fonction **res(V,F)** *:*

1. *Si V est vide, retourner F comme résultat. Sinon, choisir une variable v dans l'ensemble V des variables à éliminer.*

2. *Déterminer l'ensemble $CPos(v, F)$ des clauses qui contiennent v et l'ensemble $CNeg(v, F)$ de celles qui contiennent $\neg v$.*

3. *Calculer l'ensemble $NouvC(v,F)$ de toutes les clauses résolvantes obtenues par coupure sur v, à partir des clauses de $CPos(v,F)$ et de $CNeg(v,F)$.*

4. *Eliminer de $NouvC(v,F)$ les clauses correspondant à des tautologies.*

5. *Soit $NouvF$ la formule obtenue après élimination des clauses subsumées de l'ensemble $F \backslash (CPos(v,F) \cup CNeg(v,F)) \cup NouvC(v,F))$*

6. *Renvoyer $res(V \setminus \{v\}, NouvF)$.*

2.3 Exemple de calcul de résolvant sur un système physique simple

Cet exemple est extrait de [14] et montre l'intérêt du calcul de résolvant sur un cas simple. Il concerne la modélisation d'un tableau électrique schématisé figure 2.1.

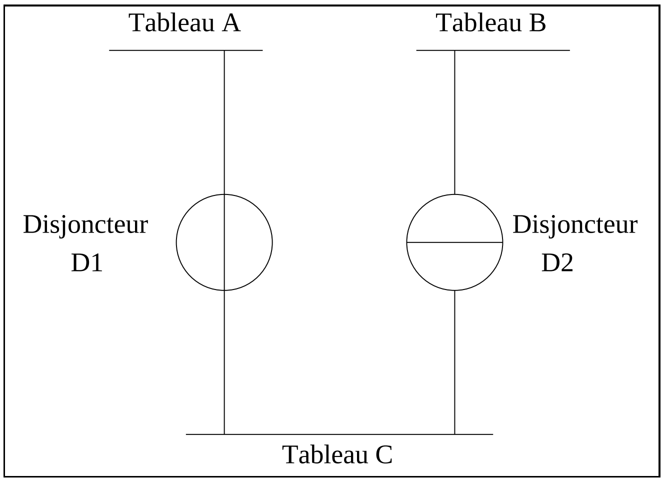

FIG. 2.1 - *Schéma d'alimentation électrique*

Les noms sont définis comme suit :

- TA représente la présence de courant sur le tableau électrique A,
- TB représente la présence de courant sur le tableau électrique B,
- TC représente la présence de courant sur le tableau électrique C,
- D1F code létat du disjonteur D1 (D1F= 1 ssi le disjoncteur est fermé),
- D2F code l'etat du disjoncteur D2.

Les variables propositionnelles qui caractérisent l'état physique de ce système sont donc {*TA, TB, TC, D1F, D2F*}, où *DiF* impose que le disjoncteur i soit fermé, et *Tx* témoigne de la présence de courant sur le tableau électrique *x*. La formule modélisant le système physique est :

$$((TA \wedge D1F) \vee (TB \wedge D2F)) \leftrightarrow TC$$

Cette formule équivaut aux six clauses suivantes :

1. $\neg TA \vee \neg D1F \vee TC$
2. $\neg TB \vee \neg D2F \vee TC$

3. $\neg TC \vee TA \vee TB$

4. $\neg TC \vee TA \vee D2F$

5. $\neg TC \vee D1F \vee TB$

6. $\neg TC \vee D1F \vee D2F$

L'élimination de $D1F$ consiste à :

- Effectuer toutes les résolutions sur $D1F$. Toutes les clauses produites, issues des clauses 1, 5 et 6 sont des tautologies. Elles sont éliminées.
- Eliminer les clauses 1,5 et 6.

L'élimination de $D2F$ consiste à :

- Effectuer toutes les résolutions sur $D2F$. Toutes les clauses produites, issues des clauses 2 et 4 sont des tautologies. Elles sont éliminées.
- Eliminer les clauses 2 et 4.

Le résolvant est alors la formule réduite à la seule clause 3, qu'on peut réécrire :

$$TC \rightarrow (TA \vee TB)$$

Cette dernière formule exprime bien une information indépendante de la position des deux disjoncteurs $D1$ et $D2$: la présence de courant sur le tableau C implique une présence de courant sur le tableau A ou sur le tableau B.

2.4 Procédure de Davis-Putnam-Logeman

De nombreuses similarités et quelques différences font du calcul de résolvant et de DPL des algorithmes très proches. Cette proximité vient du fait que tous deux sont des applications de l'expansion de Shannon. En dehors du fait que DPL procède à toutes les éliminations des variables de la formule, la différence entre les deux approches vient essentiellement de l'ordre dans lequel on décompose la formule. DPL, l'algorithme de référence pour la résolution du problème fondamental SAT, explore successivement les deux possibilités (v ou $\neg v$) alors que l'algorithme de calcul de résolvant explore les deux possibilités à la fois.
Comme nous ferons de nombreuses références à la procédure DPL dans la suite, on rappelle ici son principe, sous la forme d'un algorithme de calcul. La procédure, présentée dans [18] a deux paramètres : la formule propositionnelle F dont on veut tester la satisfiabilité et l'ensemble de toutes les variables V apparaissant dans F.

Procédure **DPL(V,F) :**

- *Si* $V = \emptyset$, *renvoyer* f *(réduit à* 1 *ou* 0*).*
- *Sinon, choisir une variable* v *dans* V*,*
- *Si* $DPL(V \setminus \{v\}, F_{v \leftarrow 1})$ *renvoie* 1*, renvoyer* 1
- *Sinon, renvoyer* $DPL(V \setminus \{v\}, F_{v \leftarrow 0})$

La procédure renvoie 1 si la formule est satisfiable et 0 sinon. Intuitivement, on peut lire la différence avec *Resolvant_Shannon* dans l'interprétation du connecteur $\vee$ présent dans l'expansion de Shannon : alors que la première procédure interprète logiquement la disjonction des deux formules pour donner un résultat, DPL renvoie le résultat 1 si au moins un des deux appels récursifs renvoie 1. La sémantique du connecteur $\vee$ est utilisée dans l'algorithme pour arrêter l'exploration dès qu'une des deux formules dont on traite la disjonction est prouvée.

2.5 Particularités du calcul de résolvant

2.5.1 Théorème de prolongement

Un des points clés du calcul de résolvant peut se résumer dans le théorème suivant, établi dans [14] et dans [9], sous deux formes différentes, mais équivalentes.

THÉORÈME 1 (Prolongement) *Soit f une formule booléenne mise sous FNC et construite sur les variables V, et $f_{V'}$ le résolvant de f par rapport au sous-ensemble V' des variables de V. Alors, pour toute affectation A' des variables de V' satisfaisant $f_{V'}$, il existe au moins une affectation A des variables $(V \setminus V')$ telle que l'affectation $A \cup A'$ satisfasse f. De plus, la restriction de tout modèle de f aux variables V' est un modèle de $f_{V'}$.*

Ce théorème permet d'appliquer l'algorithme de calcul de résolvant au problème fondamental SAT, qui consiste à tester la satisfiabilité d'une formule. En effet, si on prouve la satisfiabilité (resp. l'insatisfiabilité) du résolvant d'une formule, alors l'application de ce théorème permet d'en déduire que la formule initiale est satisfiable (resp. insatisfiable). Notons aussi la remarque suivante, par application directe du théorème de prolongement,

Remarque 1 *Soit une formule booléenne f, alors tout résolvant f_R de f satisfait $f \to f_R$.*

Il est clair que si f est satisfaite par une affectation, alors f_R l'est aussi. Pour que la remarque soit fausse, il faudrait qu'une affectation satisfasse f et non f_R, ce qui est en contradiction avec le théorème de prolongement.

Remarque 2 *Le résolvant d'une formule propositionnelle par rapport à un ensemble de variables est indépendant de l'ordre d'élimination de ces variables.*

Cette dernière remarque est aussi une application directe du théorème de prolongement. La preuve se fait en raisonnant sur les modèles du résolvant. Ainsi, pour deux résolvants R_1 et R_2 obtenus avec deux ordres d'élimination différents, tout modèle de R_1 a un prolongement qui est un modèle de la formule initiale. Or, la restriction de ce prolongement aux variables de R_2 est un modèle de R_2. Donc tout modèle de R_1 est aussi un modèle de R_2 et inversement. Donc R_1 est équivalent logiquement à R_2.
Ainsi, on est assuré que le choix de la variable sur laquelle effectuer la coupure dans l'algorithme présenté en 2.2.3 n'importe pas sur le résultat obtenu.

Enfin, ce théorème permet aussi de déduire les deux propositions suivantes, permettant d'appliquer l'algorithme aux problèmes SAT et CoSAT (voir section suivante).

Proposition 1 *Une formule est satisfiable si et seulement si le résolvant sur toutes ses variables est l'ensemble vide.*

Proposition 2 *Une formule est insatisfiable si et seulement si le résolvant sur toutes ses variables est la clause vide.*

2.6 Coût du calcul de résolvant

Dans cette section, on tente d'analyser la complexité du calcul de résolvant. Pour cela, rappelons deux propositions classiques en complexité.

Proposition 3 (Cook) *Le problème SAT, répondant à la question de la satisfiabilité d'une formule propositionnelle est NP-Complet*

Proposition 4 *Le problème CoSAT, répondant à la question de la validité d'une formule propositionnelle est CoNP-Complet.*

Il en résulte assez facilement que le problème étudié, qui répond aux deux questions précédentes selon que le résolvant sur toutes les variables est vide ou qu'il n'a que la clause vide, est clairement NP-Difficile et CoNP-Difficile. On peut donc s'attendre à un problème plus *difficile* que SAT, en terme de traitement pratique.

2.6.1 Liens avec le calcul d'impliqués premiers

On a vu que parmi les applications du théorème de prolongement, le résolvant d'une formule est impliqué logiquement par cette formule. Or, un des problèmes que l'on peut trouver régulièrement dans la littérature et qui s'apparente le plus au calcul de résolvant, est certainement celui du calcul d'impliqués premiers. Pour mieux comprendre les liens qui unissent ces deux problèmes, nous tentons ici de faire un parallèle entre eux, qui devrait intuitivement montrer que le calcul de résolvant, plus dur que SAT, semble plus facile que le calcul d'impliqués premiers.

On appelle *impliqués premiers* d'une formule propositionnelle f l'ensemble des clauses c telles que :

- $f \models c$
- Pour toute clause c', si $f \models c'$ et $c' \models c$, alors $c \models c'$

On établit ici un lien entre les impliqués premiers et le calcul de résolvant, par l'énoncé du théorème suivant.

THÉORÈME 2 *Soit C l'ensemble des impliqués premiers, sous forme normale conjonctive, d'une formule f construite sur un ensemble V de variables propositionnelles. Alors $C_{p(V_e)}$, conjonction de toutes les clauses de C ne contenant aucune des variables de V_e, est une formule équivalente au résolvant de f sur $V \setminus V_e$.*

Preuve : la preuve est donnée en Annexe A.1, page (VI).

Même si le calcul de résolvant ne semble pas apparaitre dans la littérature, de nombreux travaux portent sur le calcul d'impliquants et d'impliqués premiers. La méthode traditionnelle de calcul d'impliqués premiers et un lien entre les impliquants premiers et les impliqués premiers peuvent être trouvés dans [17]. On peut tout de même exprimer de façon simple ces liens en notant que les impliqués d'une formule f sont les impliquants de sa négation $(\neg f)$ pour lesquels on change les $\vee$ en $\wedge$, les $\wedge$ en $\vee$ et les littéraux en leurs opposés.

2.7 Liens avec la Directional Resolution

Depuis 1994, les travaux d'Irina Rish et de Rina Dechter ont tenté de revisiter la procédure originale de Davis et Putnam. Dans [9], les auteurs formalisent la *Directional Resolution* grâce à un partitionnement des variables en *Buckets*. Étant donné un ordonnancement $d = Q_1, \ldots, Q_n$, le Bucket pour Q_i contient toutes les clauses avec la variable x_i et aucune des variables d'ordre supérieur. La procédure de *Directional Resolution* (DR) procède sur les buckets dans l'ordre inverse de d comme suit :
Soit le bucket Q_i, calculer toutes les résolutions possibles sur la variable d'ordre i et insérer les clauses produites dans les buckets appropriés. La différence avec notre algorithme est que DR *n'élimine pas* les clauses sur lesquelles la résolution a été effectuée et doit traiter toutes les variables. Cependant, en dehors du fait que DR élimine toutes les variables de la formule, les deux algorithmes fonctionnent exactement selon les mêmes principes.
L'approche décrite dans [9] reste orientée vers la résolution du problème SAT, les auteurs exhibant des classes de problèmes pour lesquelles DPL (voir section suivante) est inefficace en comparaison à l'algorithme initial de Davis et Putnam. Cette implémentation, la première à *redonner vie* à l'algorithme DP, reste malgré tout un très bon point de repère pour d'éventuelles comparaisons de performance entre les deux approches. En dehors de cette comparaison, nous verrons, tout au long du rapport, en quoi notre approche permet de compléter efficacement le travail de Dechter et Rish, notamment en ce qui concerne l'étendue des applications de l'algorithme initial (qui ne se limite pas au problème SAT). Nous présenterons aussi de nouvelles heuristiques pour la sélection de *bons* buckets et une structure de données adaptée aux subsomptions, que les Dechter et Rish ne semblent pas avoir utilisé. Enfin, dans les chapitres 5 et 6, nous verrons que l'on peut aller plus loin dans le calcul de résolvant, avec deux approches originales, qui pourraient bien être directement appliquées à la Directional Resolution.

2.7.1 Complexité

Dans [9], les auteurs établissent le théorème suivant, en écrivant l'ensemble des *buckets* à la fin du calcul comme $E_d(\phi)$, où ϕ est la théorie initiale :

THÉORÈME 3 (Directional Resolution Complexity) *Étant donnée une théorie ϕ et un ordre d sur ses symboles propositionnels, la complexité en temps de l'algorithme DR est en $O(n \ . \ |E_d(\phi)|^2)$, où n est le nombre de symboles propositionnels du langage.*

L'application directe de ce théorème permet d'identifier une classe de problèmes traitable par l'algorithme en temps polynomial.

Théorie de 2-CNF Pour SAT, le problème 2-CNF est P-complet. Dans le cas du calcul de résolvant, étant donné que la coupure sur deux clauses de longueur 2 donne une clause de longueur inférieure ou égale à deux, et qu'il y a au plus (n^2) clauses de longueur 2 avec n variables, la théorie finale, au sens de Dechter et Rish, est donc de taille polynomiale. Le théorème précédent permet d'affirmer que l'algorithme DR s'exécute en temps polynomial.

Chapitre 3

Mise en œuvre de l'algorithme

Ce chapitre traite de l'implémentation de l'algorithme de calcul de résolvant tel que nous l'avons défini précédemment. Après une présentation des résultats obtenus avec une première version du programme, nous proposons une structure de données mieux adaptée au traitement des subsomptions, qui a donné lieu à une seconde implémentation de l'algorithme. Nous essayons ensuite de mesurer l'impact du choix des structures de données et des heuristiques sur les performances de l'algorithme. Puis, afin de cerner les points forts et les limites de l'algorithme, nous l'avons expérimenté sur de nombreux exemples fournis par ÉdF ainsi que sur des séries de problèmes générés aléatoirement. D'autres expérimentations sont présentées dans le chapitre suivant.

3.1 Notes sur la technique expérimentale utilisée

Pour bien faire, dans le cadre d'expérimentations sur des exemples aléatoires, il faudrait pouvoir faire des moyennes sur plusieurs centaines d'exécutions, pour une même classe d'exemple. Malheureusement, nous avons dû tenter de mêler la rigueur d'une telle expérimentation avec les contraintes temporelles d'un stage de DEA. Cependant, pour suivre la méthodologie expérimentale proposée dans [16], tous nos tests sur des exemples aléatoires sont donnés avec le nombre d'expérimentations correspondantes, et tout point caractéristique que l'on a pu observer lors de ces tests est dûment analysé et commenté.

L'implémentation que l'on présente a été faite en *caml*, un langage fonctionnel permettant quelques écarts impératifs (comme la gestion de pointeurs). Même si l'utilisation d'un langage comme celui-ci a permis de finir la maquette assez rapidement, on a perdu un facteur temps important lors des exécutions. On peut estimer ce facteur de 10 à 20 selon le type d'exécution par rapport aux performances que l'on pourrait attendre d'une implémentation en C. Cette considération est importante pour relativiser les résultats obtenus.

3.2 Une première implémentation

Lors d'une première version de l'implémentation de l'algorithme, nous avons choisi des structures de données facilitant les opérations de coupure. Ainsi, nous avons été motivés par :

- Un accès rapide aux ensembles de clauses où apparaissent les littéraux sur lesquels on effectue la coupure, pour trouver au plus vite les ensembles concernés par la distribution des clauses. Pour cela, chaque variable est placée dans un tableau associant la liste des clauses où elle apparaît positivement et la liste des clauses où elle apparaît négativement. Pour faciliter cette représentation, on code les variables propositionnelles par des entiers.
- Un calcul rapide d'union d'ensemble de variables propositionnelles, pour aider le calcul des clauses produites par coupure. Les ensembles (que nous implémentons naturellement par des listes) de variables sont donc ordonnés suivant l'ordre usuel sur les entiers codant les variables ;

– Un test de tautologie rapide. Les variables propositionnelles d'une clause sont donc codées par deux ensembles disjoints, chacun codant les variables positives (resp. négatives) présentes dans la clause. Le test de tautologie est donc réduit au calcul de la différence entre deux listes d'entiers ordonnés.

3.2.1 Premiers enseignements

Une toute première maquette en *prolog* ([12]) implémentée pour cerner les difficultés de l'approche proposée et dans laquelle les variables étaient éliminées dans un ordre fixé à l'avance, a vite mis en évidence la nécessité d'heuristiques dans le choix des variables à couper. Ces heuristiques, qui constituent avec l'expérimentation de la méthode, les points clés de ce chapitre sont détaillées dans les sections suivantes. Cependant, même si ces heuristiques ont permis de traiter les exemples fournis par ÉdF, les premiers tests sur les exemples aléatoires ont montré que le programme passait énormément de temps dans les tests de subsomptions. Ceci nous a amené naturellement à choisir une structure de données plus tournée vers les tests de subsomptions.

3.3 Choix d'une structure de donnée adaptée aux subsomptions

Guidés par un soucis d'efficacité lors de la génération des clauses produites par coupures, nous n'avons pas pris soin de prendre une structure de données totalement dédiée aux subsomptions. Or, sur des exemples aléatoires de 125 clauses, on a pu compter plus de cinq cent millions de ces tests. Les tests de subsomptions, présents à l'origine pour simplifier les formules produites ont donc nécessité une structure de données mieux adaptée, en vue d'optimiser au mieux ces tests qui s'avèrent finalement très coûteux. Une des possibilité à laquelle nous avons pensé a été de prendre une structure de données permettant de mettre en commum le maximum d'information contenue dans les clauses. Le but de cette structure est de représenter un ensemble de clauses de façon arborescente, de manière à pouvoir exécuter des tests de subsomptions sur des ensembles de clauses plutôt que sur chaque clause d'un ensemble.
Étant donnée l'étroite ressemblance entre le calcul de résolvant et le calcul d'impliqués premiers, on a pu retrouver la structure de données à laquelle nous pensions dans la littérature traitant des impliqués premiers, dont le calcul doit aussi faire face à de nombreux tests de subsomptions. J. de Kleer propose une structure de données similaire dans [8], qu'il appelle *Trie*. C'est aussi le nom que nous donnerons à cette structure de données arborescente. Elle est basée sur l'observation des opérations couramment exécutées lors du calcul d'impliqués premiers d'une formule. Il observe que l'on doit très souvent :

– Tester si une clause est subsumée par un ensemble de clauses ;

– Ôter d'un ensemble de clauses toutes les clauses subsumées par une nouvelle clause ;

– Ajouter une clause à un ensemble de clauses.

Un ensemble de clauses est, comme on l'a dit, codé par un arbre. Chaque nœud de cet arbre est marqué par l'indice d'un littéral et les feuilles sont des marques de clause. Une clause est alors un chemin dans l'arbre, partant de la racine et tous les littéraux présents dans la clause sont représentés par les indices des noeuds du chemin.

Pour illustrer cette structure de données, on donne figure 3.1 le Trie correspondant à l'ensemble de clauses :

$$(x1 \vee \overline{x2} \vee x4) \;\wedge\; (\overline{x1} \vee x5) \;\wedge\; (\overline{x4} \vee x5)$$

La première branche correspond aux clauses contenant la variable comme littéral positif, la seconde branche la variable négative et la troisième branche donne le sous-arbre codant les clauses ne contenant pas la variable du nœud.

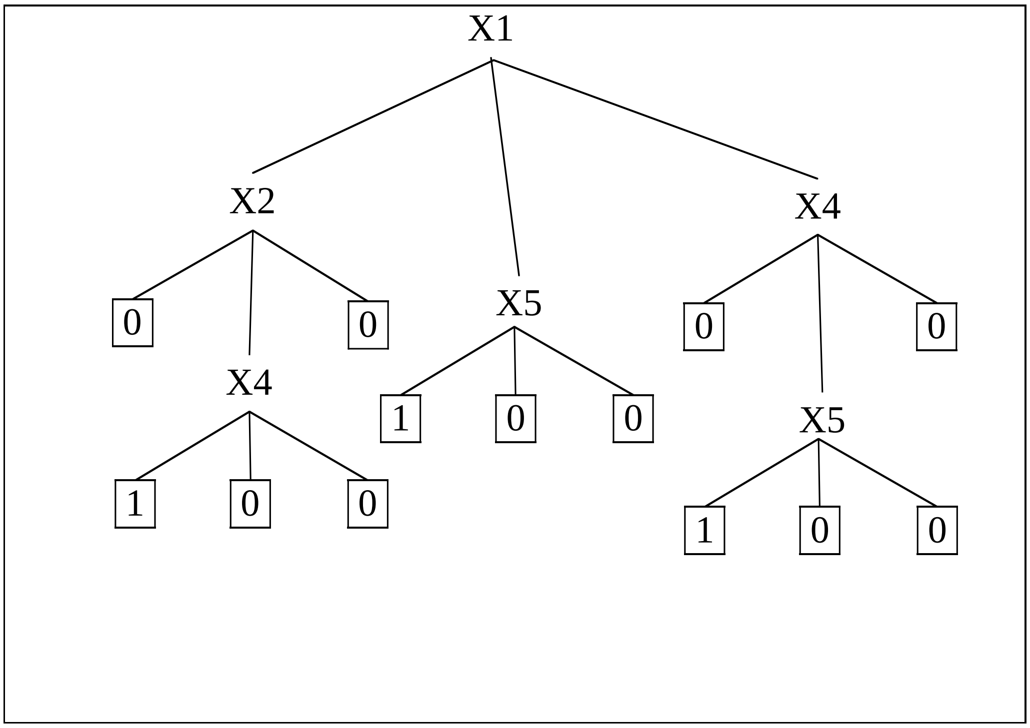

FIG. 3.1 - *Trie codant l'ensemble des clauses*

3.3.1 Algorithme sur les Tries

Pour garder un programme efficace lors des coupures, nous avons seulement ajouté un Trie contenant toutes les clauses, en gardant un lien avec toutes les structures de données introduites dans la première implémentation. L'algorithme s'écrit comme une fonction récursive où $TrieGlobal$ est le Trie codant l'ensemble des clauses de la formule :

Fonction **res(V,TrieGlobal)** *:*

1. *Si V est vide, retourner $TrieGlobal$ comme résultat. Sinon, choisir une variable v dans l'ensemble V des variables à éliminer.*
2. *Déterminer l'ensemble $CPos(v,F)$ des clauses qui contiennent v et l'ensemble $CNeg(v,F)$ de celles qui contiennent $\neg v$.*
3. *Construire le Trie $TrieNouvC(v,F)$ de toutes les clauses résolvantes obtenues par coupure sur v, à partir des clauses de $CPos(v,F)$ et de $CNeg(v,F)$.*
4. *Éliminer du Trie $TrieGlobal$ les ensembles de clauses $CPos(v,F)$ et $CNeg(v,F)$*
5. *Pour toutes les clauses C de $TrieNouvC(v,F)$, faire :*
   - *Si C n'est pas subsumée par une clause de $TrieGlobal$, ajouter C au Trie $Trie\_Global$ auquel on a oté les clauses subsumées par C.*
6. *Renvoyer $res(V \setminus \{v\}, TrieGlobal)$.*

Un tel algorithme nécessite une implémentation facilitant le calcul des ensembles $CPos$ et $CNeg$ d'après le Trie global des clauses et aidant au calcul des coupures. Ceci peut être fait avec une gestion croisée des pointeurs sur les clauses. Chaque feuille du Trie pointe sur une clause codée comme précédemment, pour faciliter le calcul de la distribution des clauses. On construit un Trie intermédiaire pour traiter au plus tôt les subsomptions entre les clauses produites.

Exemple	Temps Sans Trie	Temps Avec Trie
EDF-284	10s	15s
EDF-2210-1	87s	25s
EDF-1810	0.5s	0.6s
Chinese	1h10min	2min17s

TAB. 3.1 - *Temps d'exécution comparés avec/sans Trie*

3.3.2 Résultats sur l'introduction du Trie

On résume dans le tableau 3.1 quelques temps caractéristiques de différentes exécutions du programme avant et après l'introduction du Trie. L'heuristique utilisée est l'heuristique 3, présentée dans les sections suivantes et semblant donner les meilleurs résultats.

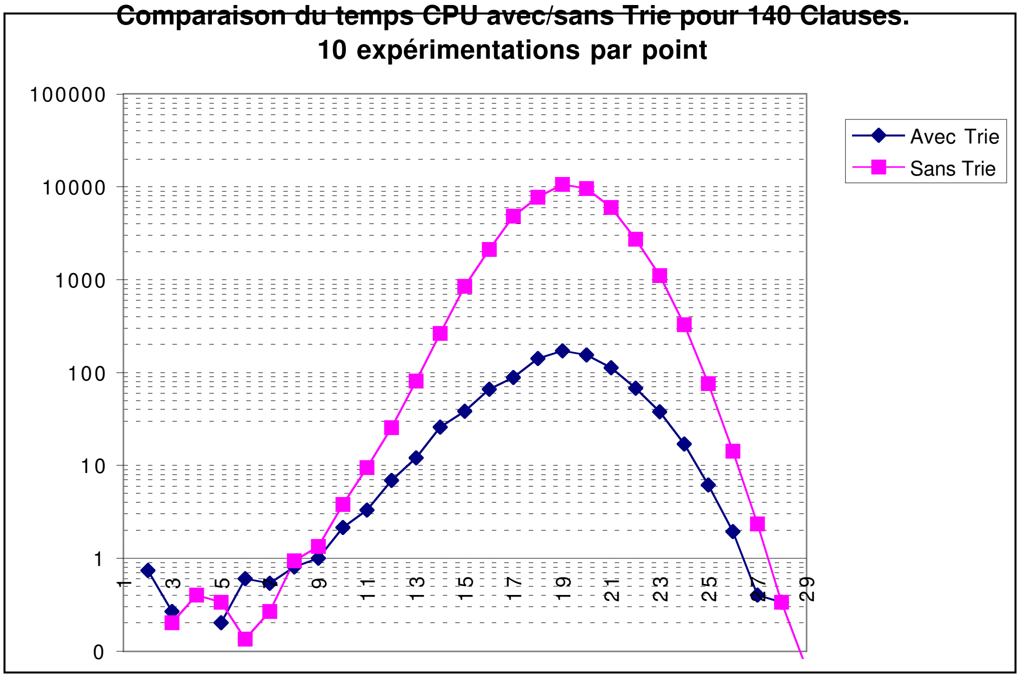

FIG. 3.2 - *Comparaison du temps par coupure nécessaire*

On peut déjà remarquer que le traitement de l'exemple *Chinese*, issu d'une modélisation de systèmes nucléaires chinois, justifie à lui seul le choix de cette structure de données. Nombre d'autres exemples, notamment dans le domaine des formules aléatoires, n'ont pu être traité qu'après l'introduction du Trie. On peut cependant être étonné par *EDF-284*, qui donne un meilleur temps avec la version sans Trie. En regardant plus précisément l'exécution, on s'aperçoit que cet exemple ne donne lieu qu'à peu de distribution de clauses lors des coupures. La gestion du Trie contenant toutes les clauses est trop coûteuse face à la simplicité de cette formule. Pour finir la comparaison entre les deux versions de l'algorithme, nous donnons, figure 3.2, les courbes de temps nécessaires par coupure sur les deux versions du programme, pour 140 clauses aléatoires *difficiles* [1] et 15 expérimentations par point. On peut lire sur les courbes des résultats de temps pratiquement similaire avec l'algorithme de Directional Resolution de Dechter et Rish en ce qui concerne le temps avec l'approche sans Trie.

1. Les exemples difficiles pour 3-SAT, formule booléenne composée uniquement de clauses de longueurs maximales 3, ont été bien identifiés dans la littérature. [6] montre qu'ils sont situés autour d'un ratio *nombre de clauses* sur *nombre de variable* de 4.28.

3.4 Les différentes heuristiques pour la coupure

Sans aucune heuristique permettant la sélection d'une variable pour la coupure, le programme a été incapable de traiter le premier exemple fourni par ÉDF, avec ou sans les Tries. Avec la plus simple des heuristiques que nous avons implémentée, le programme a pu traiter cet exemple en une dizaine de secondes. Les heuristiques que nous présentons ici peuvent compléter le travail de Dechter et Rish [10], proposant des alternatives à leur procédure de choix des variables, surtout axé sur le nombre de clauses produites.

3.4.1 Motivations

Intuitivement, deux critères principaux viennent à l'esprit :

- Premièrement, il est naturel de tenir compte du nombre de clauses gérées par le programme. Lors des tests effectués, le nombre de clauses du résolvant pouvait croître de façon exponentielle, ce qui entraînait un dépassement des capacités de la machine. Pour tenir compte de ce critère, le nombre de clauses créées lors de la coupure d'une variable sera formalisé dans le paragraphe 3.4.2. Ce nombre représente le premier critère heuristique.
- Ensuite, il faut remarquer que même si le nombre de clauses créées peut être "raisonnable", c'est la longueur des clauses produites qui peut poser problème. Une coupure entre deux clauses de longueur 3 donne une clause de longueur 4. A partir de deux clauses de longueur 4, on obtient une clause de longueur 6, et ainsi de suite, on peut produire des clauses de longueur 10, 18, 34, 66 ... Il peut donc se produire des cas où le nombre des clauses globales est stable, mais dont la longueur grandit exponentiellement. C'est sur cet autre critère qu'est basée la seconde série d'heuristiques.

3.4.2 Prise en compte du nombre de clauses créées par coupure

Nous nous intéressons ici uniquement au nombre de clauses nouvelles créées par coupure sur une variable propositionnelle. Une approximation suffisante, qui ne tient pas compte des clauses éliminées par tautologie ni de celles éliminées par la règle de subsomption, mais qui a le mérite d'être une borne supérieure de ce nombre, peut être obtenue de la façon suivante :
Soient C_k^+ et C_k^- le nombre de clauses contenant la variable propositionnelle k sous forme positive et négative. Lors de la coupure, chaque clause de C_k^+ doit être coupée avec chaque clause de C_k^-, ce qui entraîne la création de $C_k^+ * C_k^-$ nouvelles clauses. Par contre, toutes les clauses de C_k^+ et de C_k^- vont être supprimées. Le nombre total de clauses va donc varier de :

$$n_k = (C_k^+ * C_k^-) - C_k^+ - C_k^-$$

Le but de la première série d'heuristiques implémentées (1, 2 et 3) est tout simplement de trouver la variable propositionnelle permettant de minimiser n_k à chaque itération. Notons que seule cette heuristique est prise en compte dans l'implémentation de DR, dans [9].

Prise en compte de particularités

Nous allons voir que le simple calcul de n_k peut permettre de prendre en compte des cas particuliers importants, en dehors du nombre de nouvelles clauses construites.

- *Variables pures* : Une variable p_k est pure si elle n'apparaît que positivement ou négativement dans une formule. Or, affecter la valeur vraie au terme dans lequel apparaît cette variable, entraîne l'élimination des clauses contenant ce terme. Affecter la valeur faux à ce terme entraîne la simplification des clauses contenant ce terme (par suppression des clauses contenant l'opposé de ce littéral). Dans ce cas, le résolvant correspond exactement à l'ensemble obtenu lorsque l'on suppose le littéral faux. Le choix d'une variable pure provoque donc une simplification du résolvant. Il est donc naturel de privilégier de tels choix.
  L'heuristique minimisant n_k prend bien ces cas en priorité, car si n_k est pure, alors $C_k^+ = 0$ (resp.

$C_k^- = 0$), d'où, simplement, $n_k = -(C_k^-)$ (resp. $n_k = -(C_k^+)$). Donc, si il existe k tel que p_k est pure, alors n_i (i dans l'ensemble des variables candidates à la coupure) est minimum pour une variable pure ($n_i < -1$).

- *Clauses unitaires* : Il est intéressant de traiter en priorité les clauses unitaires (ne contenant qu'une variable p_k positivement (resp. négativement)) car une condition nécessaire à la satisfiabilité de la formule traitée est que $p_k = vrai$ (resp. $p_k = faux$). Dans ce cas, l'élimination de la variable p_k entraîne uniquement des simplifications dans la formule initiale (par propagation unitaire). Montrons que ce cas est bien pris en compte par le simple calcul de n_k.
  Si C_i est une clause unitaire ne contenant que la variable propositionnelle p_k, alors prouvons que $n_k = -1$. En effet, l'application des règles de subsomption implique que toute clause C_j contenant p_k positivement (resp. négativement) est éliminée car subsumée par $C_i = p_k$. On a donc $C_k^+ = 1$ d'où $n_k = (1 * C_k^-) - 1 - C_k^- = -1$ (resp. $C_i = \neg p_k$, $C_k^- = 1$ et $n_k = (C_k^+ * 1) - C_k^+ - 1 = -1$).
  Comme l'heuristique minimise n_k, toute clause unitaire est coupée en priorité, si il n'existe pas de variable pure.

- *Variables unitaires* : Une variable p_k est unitaire si et seulement si elle n'apparaît que dans une seule clause positivement ou négativement et un nombre quelconque (> 0) de fois sous l'autre forme. Alors, on a $n_k = -1$, pour les mêmes raisons que pour les clauses unitaires.

Dans la suite, étant donnée une fonction heuristique H et une variable propositionnelle k, nous notons par H_k la valeur de $H(k)$.

Première heuristique

La première heuristique se fonde uniquement sur la valeur de n_k :

$$H1_k = n_k$$

Elle sélectionne la variable candidate à la coupure qui minimise le nombre de clauses créées.

Ce premier critère heuristique, qui a l'avantage d'être d'une grande simplicité, ne permet pas de départager les variables unitaires, pour lesquels on a toujours $H_k = -1$. Cependant, si le nombre total de clauses après coupure reste le même pour toutes les variables unitaires, le nombre de clauses affectées par ce choix correspond à C_k^- (ou C_k^+). Or cette modification n'est pas forcément sans effets sur la longueur des clauses produites : si on coupe deux clauses de longueur a et b, alors la clause résolvante est de longueur $a + b - 2$. Les deux heuristiques suivantes essayent de prendre en compte ce critère.

Amélioration : Les heuristiques 2 et 3

Les heuristiques 2 et 3 ont pour objectif de guider plus finement la sélection dans le cas où il y a plusieurs variables unitaires, en prenant en compte le nombre des nouvelles clauses produites par coupure. Deux options sont possibles :

- Si on maximise le nombre de clauses touchées par la coupure sur p_k, alors on a plus de chance de tomber sur des clauses qui sont des tautologies (chaque clause contient plus de littéraux donc on a plus de chance de trouver deux littéraux commun, au signe près), ou qui sont subsumées par d'autres clauses. Mais ces avantages sont bien faibles en regard de l'augmentation significative des clauses de longueur plus grande.

- Si on minimise le nombre de clauses touchées par la coupure sur p_k, on diminue le nombre de clauses longues du problème. On prend ainsi une option sur une suite du calcul plus facile.

Dans les deux heuristiques qui suivent, on cherche à maximiser ou à minimiser C_k^- (resp. C_k^+). La seule information du calcul de n_k ne suffit plus. Il faut d'abord découper l'ensemble $\mathcal{A}$ des variables propositionnelles candidates à la coupure en trois sous-ensembles (variables pures, unitaires et autres), pour pouvoir garder un ordre global sur *l'importance* des variables. On dit qu'une variable p_k dont le coût H_k calculé par l'heuristique est minimum est la plus importante.

On appelle donc $\mathcal{P}$ l'ensemble des variables pures, $\mathcal{U}$ l'ensemble des variables unitaires et enfin $\mathcal{Z}$ l'ensemble des variables restantes, soit, formellement :

$$\mathcal{P} = \{p_k \in \mathcal{A} \ / \ (C_k^+ = 0 \ ou \ C_k^- = 0)\},$$

$$\mathcal{U} = \{p_k \in (\mathcal{A} \setminus \mathcal{P}) \ /(C_k^+ = 1 \ ou \ C_k^- = 1)\},$$

$$\mathcal{Z} = \mathcal{A} \setminus \mathcal{P} \setminus \mathcal{U}$$

Il faut maintenant établir un classement de l'importance de chacune des variables de $\mathcal{A}$. On a vu qu'il était plus intéressant de couper sur une variable pure. Donc si H_k est le coût calculé par l'heuristique, alors pour tout p_i dans $\mathcal{P}$ et tout p_j dans $\mathcal{U} \cup \mathcal{Z}$, on doit avoir $H_i < H_j$.
De même, il est plus intéressant de couper sur une variable unitaire que sur une variable quelconque de $\mathcal{Z}$ (le nombre total des clauses est alors réduit de 1). Pour satisfaire cela, posons que pour tout p_i dans $\mathcal{U}$ et tout p_j dans $\mathcal{Z}$, on doit avoir $Cout_i < Cout_j$.
En résumé,

$$\forall i \in \mathcal{P},\ \forall j \in (\mathcal{U} \cup \mathcal{Z}),\ H_i < H_j$$

$$\forall i \in \mathcal{U},\ \forall j \in \mathcal{Z},\ H_i < H_j$$

Raffinements

Maintenant que cet ordre global est établi, on peut ordonner les variables propositionnelles dans chacun des sous-ensembles, sans affecter l'ordre d'importance des variables entre les ensembles.

- Pour toute variable de $\mathcal{Z}$, on cherche simplement à minimiser le nombre des clauses résolvantes. Dans ce cas, on peut reprendre le calcul de n_k et imposer que :

  $$\forall i, j \in \mathcal{Z},\ n_i < n_j \Leftrightarrow H_i < H_j$$

- Pour toute variable de $\mathcal{U}$, on pose d'abord

  $$max_k = C_k^+ + C_k^-$$

  Alors, si on maximise (resp. minimise) max_k, on maximise (resp. minimise) le nombre de clauses rallongées par coupure sur k. On obtient bien des clauses rallongées car une variable peut apparaître une seule fois sous une forme, mais dans une clause de longueur quelconque, mis à part le cas des clauses unitaires.
  Par exemple, si p_k apparaît dans i clauses de longueur 4 sous forme positive et dans une seule clause de longueur 8 sous forme négative, alors couper sur cette variable entraîne l'élimination des $(i+1)$ clauses citées et la création de i clauses de longueur $(8+4) - 2 = 10$.
  On peut donc tirer de cela deux sous-heuristiques,

  1. L'heuristique 2, qui maximise les chances de trouver des tautologies parmi les clauses résolvantes, cherche à maximiser max_k, soit, formellement :

     $$\forall i, j \in \mathcal{U},\ max_i > max_j \Leftrightarrow H_i < H_j$$

2. L'heuristique 3, qui minimise le nombre de clauses longues du résolvant (minimise max_k), pose :

$$\forall i, j \in \mathcal{U}, max_i > max_j \Leftrightarrow H_i > H_j$$

– Enfin, on peut se demander s'il est bien nécessaire d'imposer un ordre d'importance sur les variables de $\mathcal{P}$. Si on a le choix entre deux variables pures p_i et p_j à l'itération k, éliminer une des deux variables ne change pas la pureté de l'autre. Donc si on choisit p_i, on choisira plus tard p_j en priorité et vice-versa. On peut donc admettre que toute variable de $\mathcal{P}$ peut, sans perte d'efficacité, répondre à :

$$\forall i, j \in \mathcal{P}, H_i = H_j$$

3.4.3 Prise en compte de la longueur des clauses résolvantes

Au cours de l'expérimentation des heuristiques précédentes sur des exemples aléatoires et concrets, les formules entraînant une grande difficulté étaient celles qui :

– généraient un grand nombre de clauses résolvantes à chaque coupure (croissance exponentielle)

– généraient la création de clauses de plus en plus longues.

A partir de ces deux observations, un nouveau critère heuristique a été établi, mis en oeuvre au travers d'une quatrième fonction heuristique, qui essaye d'estimer la longueur moyenne des clauses du résolvant, en vue de le minimiser. Si le coût en calcul d'un tel critère peut, au premier abord, sembler important, il semble néenmoins représenter un bon compromis entre information fournie et coût de la fonction heuristique. Montrons tout de même que ce coût peut être "raisonnable".

Soit L_k^+ (resp. L_k^-) la liste des longueur des clauses contenant la variable propositionnelle k sous forme positive (resp. négative), posons d'abord S_k la longueur totale de toutes les clauses résolvantes, on a :

$$S_k = \sum_{i \in L_k^-,\ j \in L_k^+,\ i \neq j} (i + j) - 2,$$

alors la moyenne des longueurs des clauses construites par coupure sur la variable k est :

$$M_k = \frac{S_k}{m * n}$$

avec

$$m = Longueur(L_k^+)$$

$$n = Longueur(L_k^-)$$

Posons ensuite

$$L_k^+ = [a_1, a_2, ..., a_m] \ et \ L_k^- = [b_1, ..., b_n]$$

Il vient

$$S_k = (a_1 + b_1) + (a_1 + b_2) + (...) + (a_1 + b_n) + (...) + (a_m + b_1) + (a_m + b_n) - 2(m * n)$$

qu'on peut aussi écrire

$$S_k = n. \sum_{i=1}^{m} a_i + m. \sum_{i=1}^{n} b_i - 2(m * n)$$

Si on suppose que les listes des longueurs des clauses de L_k^+ et L_k^- sont mises à jour en même temps que les deux listes L_k^+ et L_k^- alors le coût de cette heuristique devient en $o(f)$ où $f = m + n$ est la fréquence de p_k (simple parcours des listes L_k^+ et L_k^-).

Autres critères possibles sur la longueur des clauses

Il peut être intéressant de prendre en compte non pas la longueur moyenne des clauses du résolvant, mais leur longueur maximale ou minimale. Malheureusement, en raison du manque de temps, aucun test n'a pu être fait dans cette direction. Mais on voit bien que les considérations précédentes restent valides en ce qui concerne le coût d'une telle heuristique, dans la mesure où sommer des listes ne coûte pas plus cher (au sens complexité) que la recherche d'un minimum ou d'un maximum dans deux listes non triées.

On aimerait aussi pouvoir dire qu'une clause de longueur 4 est plus intéressante que 10 clauses de longueur 3. Ceci ne peut être pris en compte ni par une moyenne, ni par un maximum ou un minimum sur la longueur des clauses. En fait, pour prendre en compte ce cas précis, il suffit par exemple de calculer la somme des longueurs des clauses (dans notre exemple, 4 est inférieur à 10*3=30).

L'heuristique 5 comme synthèse des précédentes

La dernière heuristique possible, qui pourrait prendre en compte une synthèse des critères de choix présentés jusque là, se base d'abord sur l'heuristique 3 pour le choix des variables propositionnelles de $(\mathcal{P} \cup \mathcal{U})$. Ensuite, elle implémente la remarque précédente : sommer la longueur de toutes les clauses créées par coupure sur une variable. Ce dernier critère heuristique représente le compromis entre longueur des clauses et nombre de clauses. L'intérêt d'une telle implémentation et des tests qui suivent est que l'idée de cette heuristique n'est pas naturelle. On voit bien comment elle fonctionne, mais seuls les tests sur des exemples aléatoires prouvent qu'elle semble être un bon choix parmi les 5 heuristiques que nous avons implémentées. La différence avec les autres heuristiques vient donc du choix de la variable candidate à la coupure dans le cas d'un choix dans l'ensemble des variables de $\mathcal{Z}$: on choisit la variable qui minimise la somme des longueurs des clauses du résolvant. Les considérations précédentes concernant le coût d'une telle heuristique restent valides.

Autres heuristiques Nous avons aussi implémenté d'autres heuristiques particulières que nous ne détaillons pas dans ce rapport. Par exemple, en voyant que le programme butait toujours à la fin du calcul sur certaines variables qui géneraient des coupures pratiquement intraitables, une heuristique a consisté à prendre les k variables les plus *difficiles* (celles qui apparaissaient à la fin) et à relancer la totalité du calcul en éliminant ces variables en premier. Cette heuristique a pu augmenter le nombre de coupures réalisables sur des exemples comme *baobab*, le faisant passer à deux coupures en aval. Mais nous n'avons pas eu d'autres résultats significatifs. Cependant, encore une fois, il serait intéressant de poursuivre plus en avant le travail dans cet axe, en vue de mieux comprende le rôle des variables *difficiles* dans les exemples intraitables.

On voit enfin que nombre d'heuristiques peuvent être implémentées pour la sélection d'une variable candidate à la coupure. Celles-ci sont testées dans les sections suivantes et leur présentation pourrait compléter le travail de Dechter et Rish.

3.5 Résultats expérimentaux

Avant de mener un étude sur les heuristiques à la fois dans le cas de problèmes concrets et aéatoires, nous menons une petite étude des performances de l'algorithme sur le problème initial du traitement d'arbres de défaillance.

3.5.1 Problèmes concrets

Les exemples traités auxquels on fait réféference dans le tableau 3.2 sont tous issus de la génération de modèles de systèmes physiques. Ces problèmes sont des arbres de défaillance (on en trouvera une introduction en 4.1) qui génèrent la plupart du temps un grand nombre d'impliquants premiers. Lorsque nous mentionnons *impossible* dans le tableau, c'est que soit les coupures entraînent une explosion de la consommation mémoire, soit que le programme n'a pas abouti en plus de 3 heures de calcul. Nos tests ont été effectués sur des Ultra Sparc 2 avec 64 Mo de mémoire avec l'heuristique 3. Les temps de comparaison avec *Aralia*, qui est un des meilleurs outils de vérification d'arbre de défaillance actuel, sont issus de [21], calculés sur un Ultra Sparc 1 avec 500 Mo de mémoire.

Exemples	Nombre de clauses/variables	Nombre de variables à éliminer	Temps Aralia	Temps Calcul de résolvant	Nombre de clauses du résolvant
ftr10	461c/269v	94	Non Connu	16s	305
baobab2d	218c/97v	65	0.1	553s	4805
baobab3d	380c/187v	107	6s	Impossible	16200
das9204b	130c/83v	30	0.2s	Impossible	16704
chinese	129c/61v	36	<0.1s	129s	392

TAB. 3.2 - *Temps de traitement de certains exemples concrets issus du domaine de la fiabilité*

La lecture de ce tableau peut donner l'impression, à juste titre, que le calcul de résolvant n'est pas adapté au problème de la vérification d'arbres de défaillances. Ceci est principalement dû au fait qu'il existe d'autres méthodes fondamentalement différentes, qui ne sont pas directement limitées par le nombre de clauses du résolvant. *Aralia*, par exemple, a été optimisé pour son application sur des exemples concrets issus de modélisation de systèmes physiques qui, à la différence des exemples aléatoires, possèdent une expansion de Shannon assez facilement calculable. Ces algorithmes sont basés sur l'expansion de Shannon de la formule traitée et procèdent en fait à des substitutions des variables à éliminer par des sous-formules équivalentes. Mais ils sont spécialisés dans ce type de traitement et entraînent une restriction énorme sur la formule de départ. Or, nous verrons dans le chapitre suivant comment l'algorithme de calcul de résolvant, initialement utilisé pour son application sur des arbres de défaillances, peut être utilisé sur nombre d'autres problèmes pour lesquels on ne peut pas compter sur des programmes comme *Aralia*, parce qu'il n'existe aucun moyen de réécrire en temps raisonnable les formules grâce à l'expansion de Shannon.

De manière assez intuitive, on peut arriver à caracteriser deux classes d'exemples sur lesquels nous avons travaillé.

Les exemples faciles On peut caractériser les exemples traitables par le calcul de résolvant simplement par le fait qu'ils n'ont pas beaucoup de clauses résolvantes (aux alentours d'un maximum de 5000 clauses pour un traitement rapide et 40000 clauses comme limite de capacité mémoire). Ceci est du au fait qu'on représente explicitement les ensembles de clauses.

Les exemples difficiles Par contre, une différence flagrante est mise à jour face aux méthodes spécialisées de traitement d'arbre de défaillance. Dans ce cas, les exemples que ces méthodes arrivent à traiter peuvent contenir des résolvants de l'ordre de plusieurs centaines de milliers de clauses. On voit bien que même en améliorant de manière significative les heuristiques que nous proposons, ces chiffres montrent que rien ne peut être fait pour limiter l'explosion du nombre de clause.

Résultats

Tests des heuristiques Sur le tableau 3.3, nous pouvons d'abord noter que certains exemples sont traités exactement dans le même temps, pour toutes les heuristiques. Ceci nous enseigne deux choses :

- Il existe des exemples pour lesquels toutes les heuristiques donnent exactement le même ordre d'élimination des variables.
- Certains exemples sont traités exactement dans le même temps. Le temps de calcul des différentes heuristiques est donc globalement le même pour toutes, de la plus simple à la plus compliquée sur ces exemples. On peut donc imaginer améliorer ses heuristiques, quitte à perdre du temps de calcul pour trouver de meilleurs choix.

Exemples	Temps H1	Temps H2	Temps H3	Temps H4	Temps H5
Chinese	129	129	129	129	129
Parity8	963	1037	**941**	1119	1099
ssa7552-038	478	490	**439**	630	612
ssa7552-158	258	258	**218**	352	341
ssa0432-003	45	37	**33**	50	47
ssa2670-130	576	592	**570**	851	828
bf1355-075	3414	**2453**	2494	2907	2865
bf2670-001	486	461	**460**	660	613
isp9603b	163	181	190	163	**160**

TAB. 3.3 - *Temps de traitement de divers exemples pour chaque heuristique*

- Il n'y a pas une seule heuristique qui donne systématiquement les meilleurs résultats, même si l'heuristique 3 est toujours bien placée. On peut donc penser qu'il reste des études à mener dans la comparaison des heuristiques et dans l'implémentation d'une heuristique regroupant les points forts que l'on peut trouver dans chacune d'elles.
- Les heuristiques prenant en compte la longueur totale des clauses résolvantes ne semblent pas donner de bons résultats face aux heuristiques plus simples prenant en compte uniquement le nombre de clauses produites par coupure.
- il serait intéressant de comprendre pourquoi c'est l'heuristique 2 qui dépasse de manière significative l'heuristique 1 sur l'exemple $bf1355-075$, alors qu'on pouvait penser que cette heuristique était inadaptée et produisait beaucoup de clauses.

3.5.2 Problèmes aléatoires

Même si on peut se demander si expérimenter l'implémentation de l'algorithme sur des exemples aléatoires permettra réellement d'améliorer les résultats sur des exemples concrets, le test sur des formules aléatoires reste cependant un passage quasi-traditionnel dans l'expérimentation des algorithmes. Non seulement de tels tests permettent un nombre illimité d'exécutions, mais certains auteurs [6] ont pu déterminer des caractéristiques de problèmes difficiles pour 3-SAT sur des exemples aléatoires, ceux dont le rapport *Nombre de clauses* sur *Nombre de variables* était proche de 4.28. D'autre part, une récente étude sur le calcul d'impliquants premiers de problèmes aléatoires a été faite dans [22] et a permis de trouver des caractéristiques sur les impliquants premiers de ces formules. Cependant, nous sommes conscients de la très grande particularité des exemples aléatoires, tous basés sur une distribution complètement uniforme des variables les composant.

Génération des exemples aléatoires

Il semble que le niveau de difficulté des exemples aléatoires soit lié à leur mode de génération. C'est pourquoi le moteur de génération d'exemples aléatoires développé est donné ici.
La procédure de génération fournit un fichier standard, sous forme normale conjonctive. Elle attend plusieurs paramètres : le nombre *nbclause* de clauses, le nombre *nbvar* de variables propositionnelles, la longueur maximale et minimale des clauses (*max* et *min*) et enfin le nombre *nbelim* de variables à éliminer par le calcul du résolvant.

- Pour forcer chaque variable à apparaître au moins une fois dans la formule, on force les n ($avec\ n \leq nbvar$) premiers tirages aléatoires à prendre les n variables propositionnelles. Le vrai tirage aléatoire des variables commence un fois que toutes les variables sont au moins présentes une fois dans la formule.
- La négation de chaque variable est faite avec la probabilité 0.5.

- La longueur de chaque clause est tirée aléatoirement au début du calcul, en fonction des longueurs maximales et minimales des clauses désirées.

- On empêche toute clause de contenir deux fois la même variable (donc pas de tautologies ni de clauses plus petites que la longueur tirée aléatoirement).

- Comme la distribution des *nbvar* premières variables n'est pas aléatoire, on mélange toutes les variables à la fin du calcul. Pour cela, on tire deux clauses au hasard puis une position dans chaque clause et on échange les variables correspondantes à ces deux positions, pour autant que cet échange ne crée pas de tautologies ni de clauses raccourcies. Ce mélange est effectué un nombre arbitraire de fois, dépendant de *nbvar*, *nbclause* et *max* (dans l'implémentation, on a fixé : nombre de mélanges = 4 * nbvar * nbclause * max).

Étude de la transition de phase pour 3-SAT

Le problème 3-SAT sur des formules aléatoires a fait l'objet de recherches intensives ayant permis de traiter des formules de plus en plus grandes, plus particulièrement ces dix dernières années. Une des caractéristiques reconnue de 3-SAT est certainement la transition de phase sur la difficulté des exemples générés. En effet, le rapport *nombre de clauses* sur *nombre de variable* est critique pour la facilité des exemples. Au-delà de 6, pratiquement toutes les formules sont insatisfiables, et au-dessous de 3.5, elles sont pratiquement toutes satisfiables. Si on a pu aussi observer ce phénomène de transition dans les temps de traitement des formules, mais de manière plus diffuse, nous avons décidé d'expérimenter notre algorithme, pour voir, figure 3.3, à quelle sorte de résolvant correspond, en moyenne, un tel rapport *nombre de clauses* sur *nombre de variable*. Dans tous les exemples traités et regroupés sur la figure, les résolvants étaient sur 10% des variables propositionnelles de la formule de départ. On retrouve alors un peu le même résultat que [22], c'est à dire un résolvant avec des longueurs de clauses moyennes différentes selon le rapport. On remarque nettement des pics de fréquence différentes pour chaque type de clauses dans le résolvant, suivant le nombre de variable dans la formule. Ces résultats restent cependant préliminaires et une étude plus approfondie des types de résolvant obtenus suivant divers paramètres initiaux semble souhaitable.

De plus, on peut bien vérifier sur la figure 3.4 que dans le cas d'un rapport égal à 1, ce qui correspond à des exemples très faciles pour des algorithmes comme DPL, le temps nécessaire au calcul de résolvant est proportionnel au nombre de clauses du problème. Signalons encore une fois que dans toute la suite du rapport, on s'intéresse uniquement aux problèmes aléatoires dans cette zone *difficile*.

Influences des autres paramètres sur la difficulté

À la différence du calcul d'impliqués premiers, le traitement d'exemples aléatoires a nécessité de fixer le nombre de variables à éliminer par calcul de résolvant. Il nous est apparu intérressant de voir comment ce nombre pouvait influencer la difficultée des exemples. La figure 3.5 donne les caractéristiques des résolvants selon le nombre de variables à éliminer. On donne aussi, mis à l'echelle, l'ordre de grandeur du temps cpu passé au calcul complet des exemples. Ceci nous a permis de mettre en évidence un résultat assez surprenant.

Sur la figure 3.5, on peut voir plusieurs points intéressants.

1. Le temps de traitement et la difficulté ne sont pas proportionels au nombre de variables à éliminer.

2. La plupart des clauses des résolvants ont une taille de 6, indépendamment du nombre de variables à éliminer.

3. Le traitement le plus difficile se situe aux alentours de la moitié des variables à éliminer.

Le premier point, qui peut tout d'abord sembler surprenant, est intuitivement du aux heuristiques. Plus elles ont le choix, plus elles peuvent prendre des variables interréssantes. Ceci peut paraître contre-intuitif parce qu'on aurait pu penser que dans les exemples aléatoires, toutes les variables jouaient le même rôle, et éliminer un plus grand nombre de variables aurait obligatoirement conduit à une plus grande complexité du calcul. Cela n'est pas le cas.

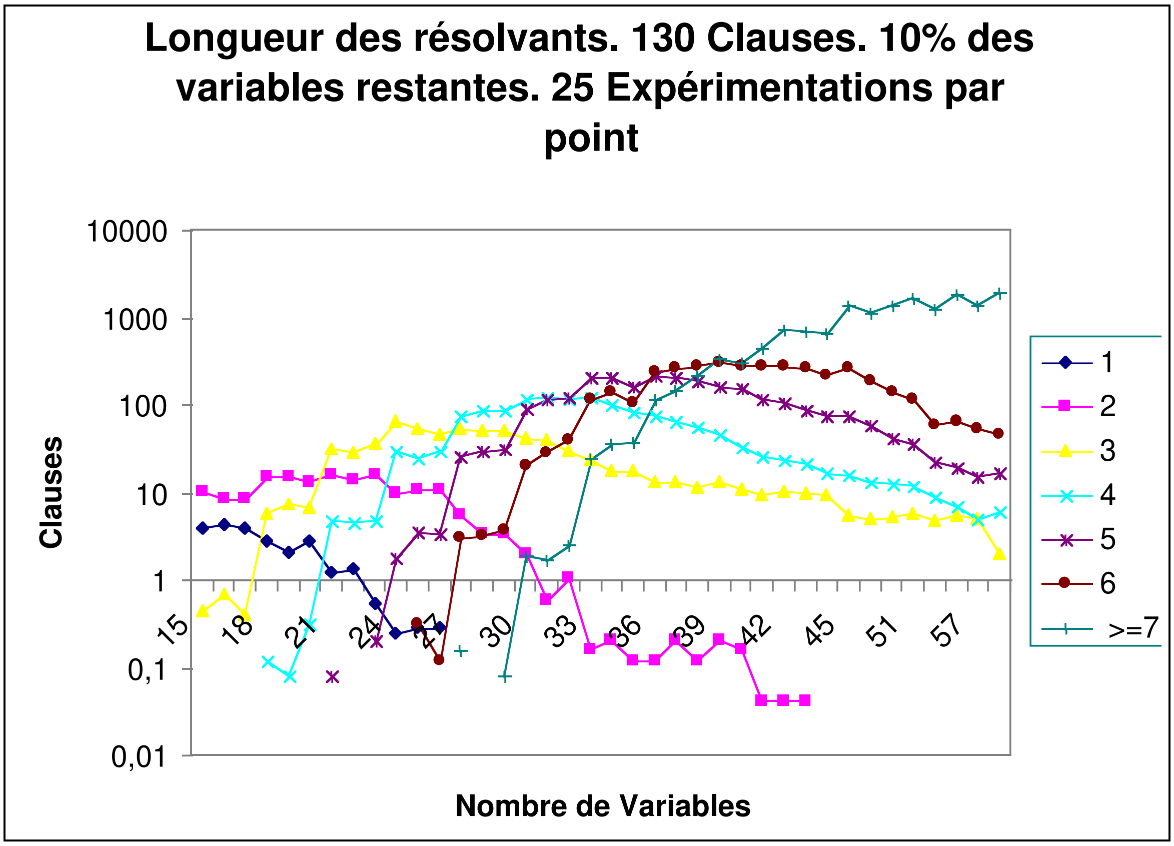

FIG. 3.3 - *Étude de la transition de phase pour 3-SAT*

Résultats

Avec l'approche classique, nous avons pu traiter des formules jusquà 200 clauses et 47 variables, ce qui dépasse largement les performances de DR (Directional Resolution), limités à une centaine de clauses pour un même rapport avec le nombre de variables sur les exemples aléatoires. On peut penser que ceci est principalement du au choix des structures de données comme le Trie, qui ont déjà, dans [8] démontrés leur avantages sur les structures de données plus *traditionnelles*. On donne sur la figure 3.6 l'évolution du temps sur des exemples aléatoires situés dans la zone difficile correspondant à la transition de phase, avec l'élimination de toutes les variables de la formule.
D'autre part, on donne sur la figure 3.7 l'évolution du nombre de clauses manipulées par le programme sur des exemples aléatoires, en fonction de la coupure. On peut y observer un phénomène de bosse intéressant : le programme est obligé de manipuler un grand nombre de clauses pour finalement obtenir un résolvant de plus petite taille.

Tests des heuristiques Il reste maintenant à tester les heuristiques avec des exemples aléatoires. Pour cela, nous avons choisi une classe de formules aléatoires et nous avons comparé les temps cumulés de l'exécution du programme sur 50 exemples aléatoires de 130 clauses et 30 variables, avec les heuristiques proposées.

Dans cette table on lit que l'heuristique 3 est la plus rapide. Cepedant, il aurait été bien de pousser l'expérimentation en vue de connaitre le nombre total de coupures, par exemple, que chaque heuristique produisait par le choix sur les coupures. On ne sait pas si ces différences de temps viennent du temps passé à calculer l'heuristique ou si ce sont les choix qui induisent ces différences. Toujours est-il que l'heuristique 3 se présente comme la plus rapide, que se soit sur les exemples concrets que sur les exemples aléatoires.

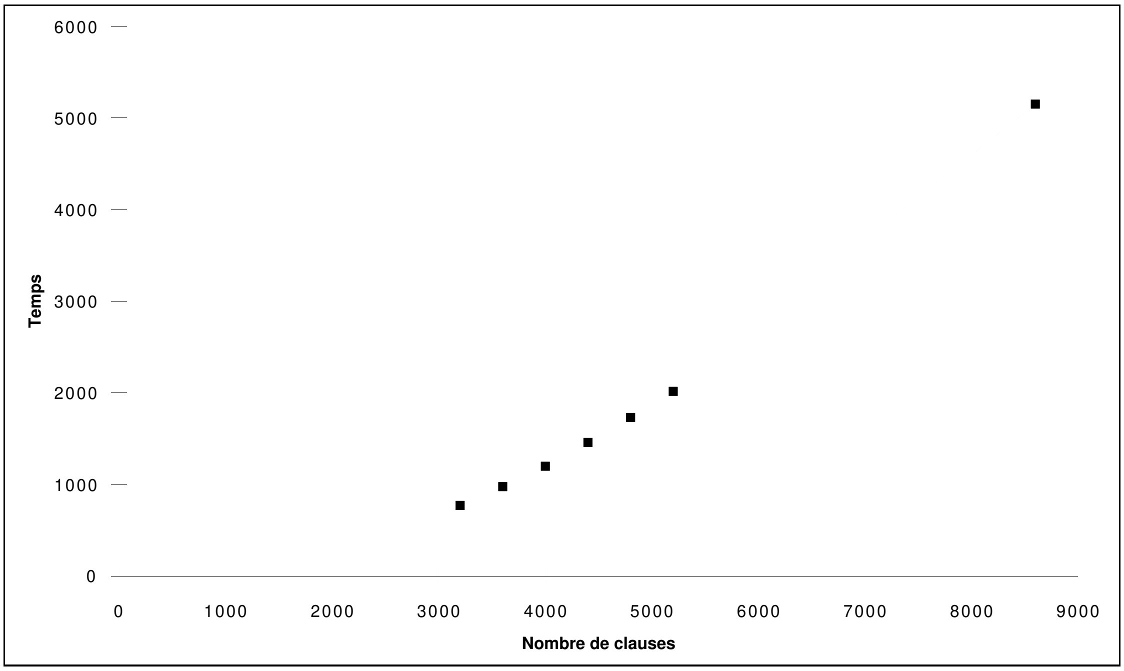

FIG. 3.4 - *Temps de calcul de résolvant dans une zone aléatoire facile*

Exemples	Temps H1	Temps H2	Temps H3	Temps H4	Temps H5
Aléatoire 130 clauses	4449	4354	**4297**	4540	4535
Aléatoire 180 clauses	6479	6438	**6437**	7024	7025

TAB. 3.4 - *Temps de traitements de divers exemples pour chaque heuristique*

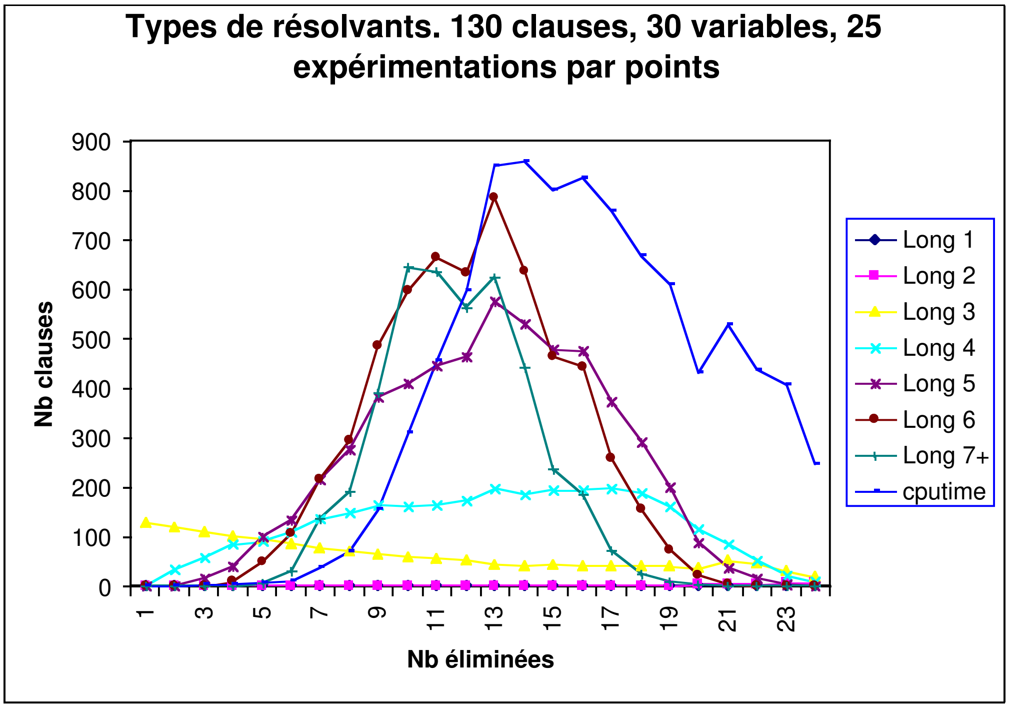

FIG. 3.5 - *Type de calcul selon le nombre de variables à éliminer*

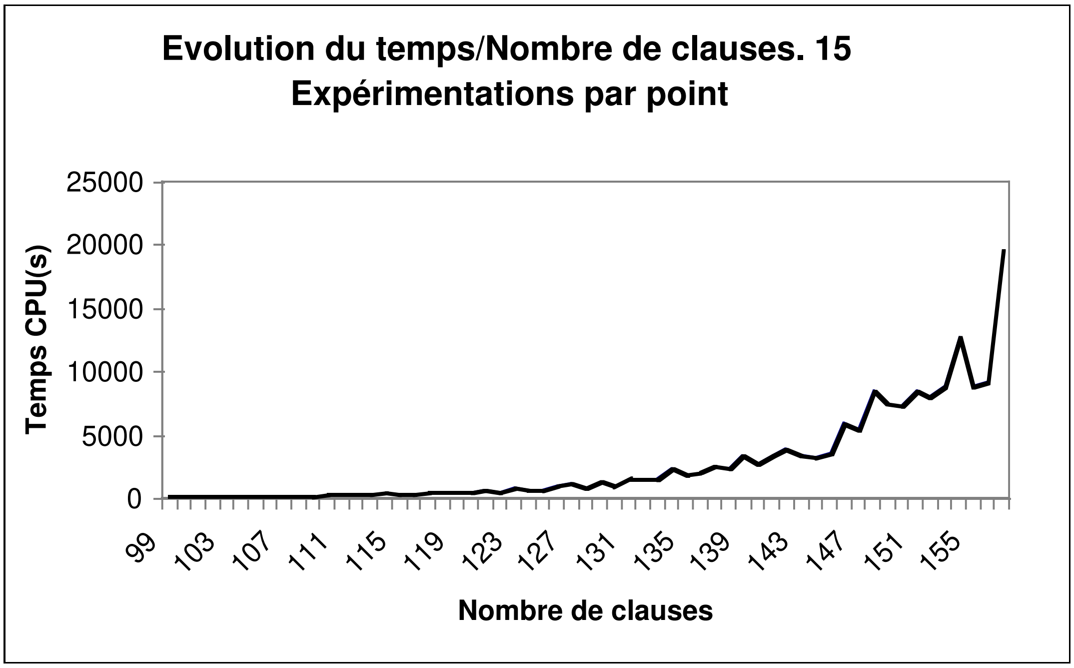

FIG. 3.6 - *Temps de calcul par rapport au nombre de clauses sur des exemples difficiles*

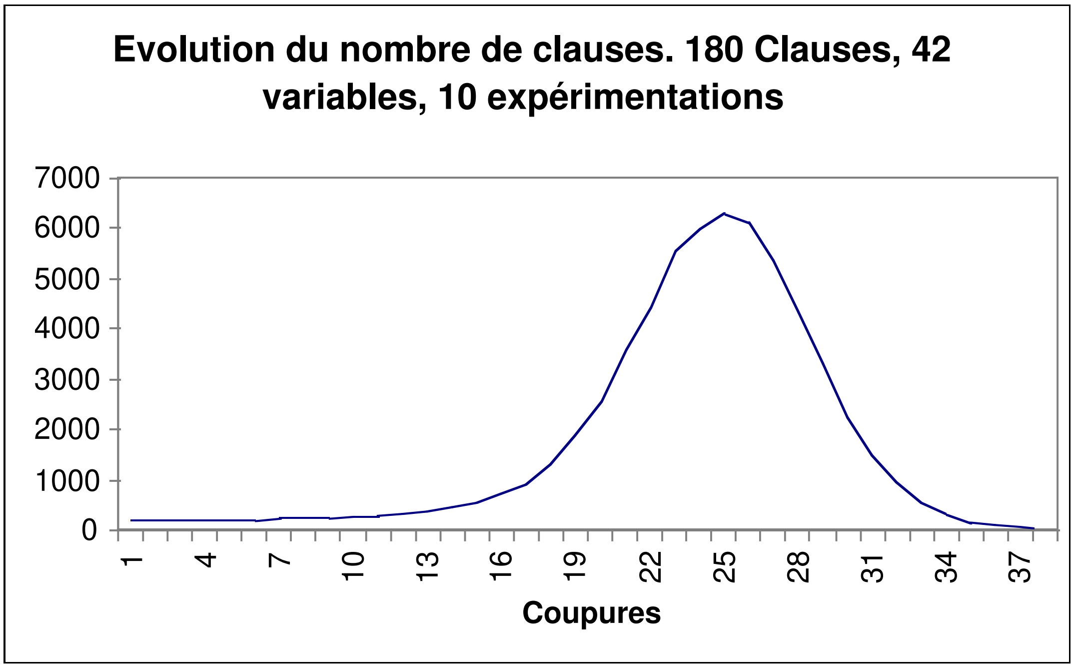

FIG. 3.7 - *Nombre de clauses manipulées pendant le calcul*

Chapitre 4

Applications du calcul de résolvant

Avant de proposer d'autres techniques de calcul de résolvant, nous présentons dans ce chapitre des applications possible de ce calcul de résolvant.

4.1 Arbres de défaillance

Due à la demande initiale d'ÉdF, l'application du calcul de résolvant aux arbres de défaillance de systèmes physiques est naturelle. Une bonne introduction aux principes des arbres de défaillances peut se trouver dans [24]. De manière simple, on peut dire qu'un arbre de défaillance est une formule propositionnelle bâtie sur un ensemble de variables représentant le bon fonctionnement d'éléments du système. Cet ensemble de variables est découpé entre les variables internes du système (variables V_I, par exemple) et les variables externes (V_E). Une variable interne (*Racine*) a une valeur sémantique particulière, représentant le mauvais fonctionnement du système tout entier pour l'affectation 1. Toute variable interne doit s'exprimer par une formule logique entre des variables internes (autre que *Racine*) ou externes. Les variables externes représentent le mauvais fonctionnement d'éléments de base du système. Par simple substitution, on peut alors exprimer le mauvais fonctionnement de tout le système par une formule logique construite uniquement sur V_E. On peut alors calculer l'ensemble des impliquants premiers de cette formule pour trouver l'ensemble des conditions minimales sur les éléments de base du système entraînant sa défaillance totale.
En prenant le lien étabi en 2.6.1 entre les impliquants et les impliqués premiers, on peut utiliser le calcul de résolvant dans les arbres de défaillance. Pour cela, il suffit de nier la variable *Racine*, puis d'utiliser le calcul de résolvant afin d'éliminer les variables internes. Ensuite, il faut appliquer l'opération que l'on vient de définir pour pouvoir interpréter les impliqués du calcul de résolvant en impliquants de la formule initiale.

Pour illustrer un arbre de défaillance, on donne figure 4.1, celui qui pourrait correspondre à une extension de l'exemple du tableau électrique donné dans le second chapitre. Celui-ci signifie que l'ampoule est éteinte (panne du système) s'il y a un perte d'alimentation sur le tableau électrique TC (TC à 1) ou si l'ampoule est grillée. On peut alors travailler sur l'arbre de défaillance en trouvant l'ensemble des conditions minimales sur les variables TB, D1, D2, TA et Ampoule grillée pour que la lampe soit éteinte. Il faut pour cela éliminer les variables internes du système, qui sont les trois variables Panne A, Panne B et TC. Le résolvant, trivial dans ce cas, est alors :

$$((TB \ ou \ D2) \ et \ (D1 \ ou \ TA)) \ ou \ (AmpouleGrillee)$$

Or, les impliqués calculés par l'algorithme de calcul de résolvant ne sont pas premiers (on peut avoir un résolvant contradictoire). Donc les impliquants ne sont pas premiers non plus, mais dans le domaine des arbres de défaillance, nombre des modèles physiques sont fait avec des variables externes apparaissant uniquement sous forme positive dans la formule. Le résolvant final est donc une formule ne contenant que des littéraux de même signe. Aucune résolution n'étant possible sur une telle formule, la minimalité est garantie par le traitement des subsomptions. Dans une telle restriction, on calcule alors exactement

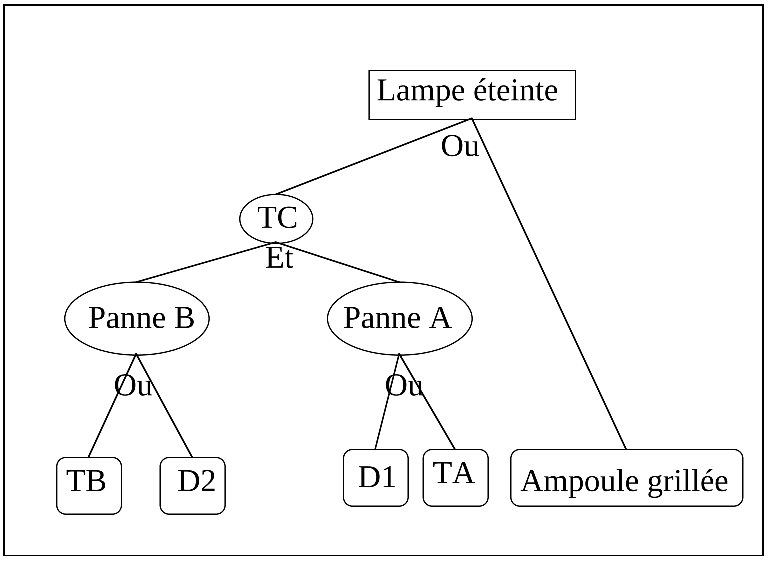

FIG. 4.1 - *Arbre de défaillance issue du tableau électrique*

l'ensemble minimal des défaillances des éléments de base du système entraînant une panne totale de ce système.

4.1.1 Comparaison avec les méthodes actuelles

Dans le cas d'une comparaison avec les meilleures méthodes actuelles ([11]), on peut voir que notre programme est largement dépassé par un système comme *Aralia*, par exemple, sur les formules codant des systèmes physiques. Les méthodes employées dans ce système sont fondamentalement différentes de celle que nous avons présenté jusque là. On verra, dans le chapitre 6 comment on peut espérer les adapter au calcul de résolvant. Intuitivement, il faut savoir que ces méthodes travaillent sur les fonctions caractéristiques des ensembles de clauses, et se détachent ainsi du nombre de clauses manipulées. Pour obtenir un algorithme efficace, cependant, ces algorithmes doivent préalablement mettre la formule initiale sous forme normale de Shannon (FNS, voir chapitre 6). Or, l'expérience montre que si cette mise sous forme normale est théoriquement exponentielle pour l'extrême majorité des formules ([25]), la restriction aux formules modélisant des systèmes physiques semble donner de bons résultats, sans toutefois que l'on ait aujourd'hui de solides justifications autres qu'expérimentales.

4.1.2 Limites intrinsèques de ces méthodes

Malheureusement, toutes les formules ne peuvent être mise sous FNS, et ces algorithmes ne peuvent clairement pas être appliqués sur des formules dont on ne sait que difficilement, encore aujourd'hui, prouver la satisfiabilité. Or, on le verra dans les sections suivantes, l'algorithme de calcul de résolvant peut être appliqué à des classes de problème de satisfiabilité et même parfois surpasser les meilleurs algorithmes systématiques mondiaux d'aujourd'hui. Utiliser ces méthodes directement entraînerait donc une restriction énorme à notre algorithme.

4.2 Compilation des bases de connaissances

L'algorithme prend une formule propositionnelle en entrée et fournit une formule propositionnelle en sortie. Dans le cas où cette formule propositionnelle est dotée d'une sémantique particulière représentant par exemple une base de connaissances, on peut obtenir une autre base de connaissances en sortie.

4.2.1 Champs de production

Dans sa thèse [23], Siegel met en évidence les utilisations de son algorithme de production dans diverses directions en IA. Celui-ci consiste à faire toutes les résolutions possibles sur une formule et à ne garder que les clauses produites correspondant à certains critères (longueur de clause, présence de littéraux...). En se référant à ses résultats, on peut dire que le calcul de résolvant correspond exactement à la production d'une formule non minimale sur un sous ensemble de ses variables propositionnelles. Malheureusement, nous n'avons pas pu comparer les résultats que nous avons obtenu avec l'algorithme proposé par [23]. Ceci reste une étude comparative intéressante à faire, puisqu'il existe déjà nombre d'applications de son algorithme de production dans diverses branches d'intelligence artificielle.

4.2.2 Étude des n-reines

Sur un exemple plus précis, avec une modélisation du problème des n-reines par une formule logique propositionnelle, il est possible de trouver des conditions liant, soit un sous-ensemble de reines, soit les reines et un sous-ensemble de cases du damier où elles peuvent figurer. Ainsi, on modélise les reines en cherchant à placer chaque reine sur une colonne. Pour simplifier, on peut déjà imposer que la reine i soit placée sur la colonne i.
Nous avons donc calculé le résolvant du problème des 6-reines sur le sous-ensemble $\{Case_{(i,1)}, \ldots, Case_{(i,6)}\}$ où $Case_{(i,j)}$ est vrai si la reine i est sur la ligne j. Nous avons obtenu des places proscrites pour les reines, places pour lesquelles on sait, par application du théorème de prolongement, qu'il n'existe pas de prolongement validant la formule. On schématise ce résultat par la figure 4.2, où les cases noires sont *interdites* pour chaque reine. La formule obtenue, par exemple en calculant le résolvant sur les variables

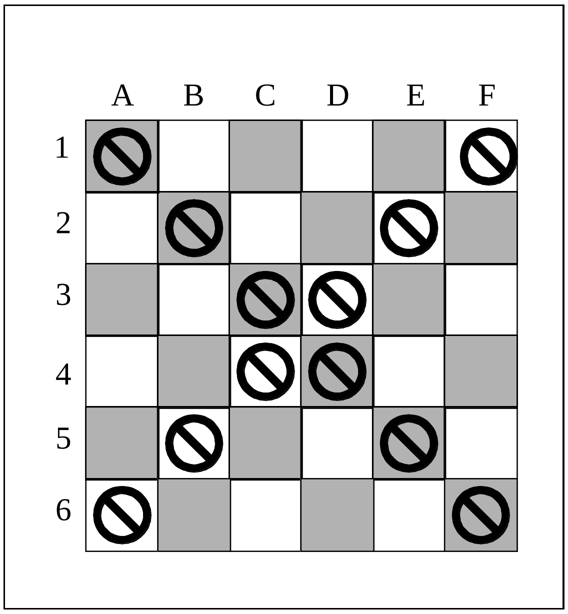

FIG. 4.2 - *Une application du résolvant sur les 6-Reines*

$\{Case_{(1,1)}, \ldots, Case_{(1,6)}\}$ est la formule contenant les deux clauses :

$$(\neg Case_{(1,1)}) \wedge (\neg Case_{(1,6)})$$

Les autres clauses de la formule signifiant qu'on doit placer la reine sur une des autres cases.

Par extension de ce principe à d'autres problèmes de satisfaction de contraintes, on peut imaginer appliquer le calcul de résolvant à des problèmes d'emploi du temps, pour ne pas trouver une seule solution, mais un ensemble de contraintes garantissant l'existence d'une solution globale. On pourrait alors imaginer de laisser la formule simplifiée entre les mains des responsables de la mise en place des emplois du temps, qui pourront ouvrir les négociations entre les participants pour trouver une solution optimale satisfaisant le plus grand nombre de personnes.

4.2.3 Transformation d'un additionneur en multiplicateur par 2

Ici, on utilise la modélisation d'un additionneur en propositionnel (modélisé par les bits a_i, b_i en entrée, s_i en sortie et r_i pour la retenue) pour imposer des restrictions sur ses entrées. Ainsi, on ajoute à la formule de départ l'équivalence entre tous les bits a_i et b_i en entrée de l'additionneur, et on calcule le résolvant pour éliminer les b_i par exemple.
On obtient une autre formule propositionnelle codant un multilplicateur par 2. Par exemple, la formule a fait apparaître la condition donnant le premier bit de sortie toujours faux (on obtient des nombres pairs).

4.3 Précompilation pour SAT

Dechter et Rish ont bien montré dans [9] l'interêt du calcul de résolvant pour le problème SAT, qui consiste à utiliser l'algorithme DP sur toutes les variables de la formule. Cependant, les auteurs n'ont pas vraiment comparé les résultats de leur algorithme sur une autre base d'exemples que celle qu'ils ont construit pour illustrer les différences entre les deux algorithmes. Notre approche, en dehors de la première partie qui expérimente aussi l'algorithme sur des exemples aléatoires, prend aussi une série d'exemples de référence issus de la littérature.

4.3.1 Sur des exemples aléatoires

Dans le cas de problèmes aléatoires, l'algorithme est particulièrement inefficace. Au contraire même, on voit nettement une augmentation de la difficulté des exemples lorsque l'on passe par une précompilation des exemples aléatoires. On peut penser que cette difficulté vient du fait qu'on a énormément travaillé sur DP avec les exemples aléatoires et que leur structure a fait l'objet d'études mathématiques et pratiques poussées. Or, une compilation de ces formules a tendance à faire disparaître l'uniformité des problèmes générés. Si nous n'avons pas poussé les investigations plus loin dans cet axe, il serait cependant intéressant de comprendre en quoi l'élimination des variables dans une formule rend cette formule plus compliquée pour être traité par un algorithme de type DPL. Enfin, on peut imaginer utiliser l'algorithme de calcul de résolvant pour générer des problèmes aléatoires *difficiles* pour DPL. Les formules alors générées n'auraient alors plus l'uniformité des formules aléatoires et pourtant aucun structure ne pourrait être identifié facilement. Générer des problèmes difficiles pour SAT peut être en soit une application intérressante du calcul de résolvant.

4.3.2 Sur des classes de benchmarks issues de DIMACS

On va voir dans la section 4.4 que l'on peut appliquer l'algorithme sur une classe de problèmes issus de DIMACS, qui répertorie des formules dont le temps de test de satisfiabilité représente un repère pour tous les algorithmes de tests de satisfiabilité mondiaux. Mais, on peut aussi utiliser le calcul de résolvant comme précompilation des formules. Ainsi, on élimine un sous-ensemble des variables et dans un certain nombre de cas que l'on a pu étudier, on obtient une formule grandement simplifiée pour un test de satifiabilité accéléré. Notons tout de même que l'identification et le calcul de résolvant en précompilation pour SAT est proposé dans les derniers algorithmes, comme C-SAT ou Tableau, qui font un calcul de résolvant simple avant de lancer DPL. Cependant, à la différence de notre approche, ce prétraitement est vite arrêté dès que le nombre de clauses résolvantes a tendance à croître ou est limité au calcul de résolvant sur des clauses de longueur 2.

Exemple(s)	Nombre de clauses/variables variables	Temps Sans Pré-compilation	Temps Calcul de résolvant	Res : Nb. Clau./Var.	Temps Avec Pré-compilation
Parity-16	3310c/1015v	2min25s	9min	7155c/76v	10s
aim-100-1_6-no-1	160c/100v	>10000s	2min6s	1088c/35v	6s
aim-100-1_6-yes-1	160c/100v	72s	6s	391c/31v	0.05s
aim-100-1_6-yes-2	160c/100v	320s	6s	307c/44v	0.15s
aim-100-2_0-no-2	200c/100v	2123s	15s	900c/58vs	0.4s
pret150-25	400c/150v	>10000s	30s	512c/24v	13s

TAB. 4.1 - *Table des temps pour la précompilation des exemples SAT*

4.4 Résolution directe de SAT : des résultats significatifs

On l'a vu dans le second chapitre, si on élimine toutes les variables, on peut appliquer le calcul de résolvant au problème SAT. Par curiosité, nous avons voulu voir dans quelles mesures le programme que nous présentons pouvait résoudre des benchmarks de ce problème. Si Dechter et Rish on bien vu que l'on pouvait sérieusement utiliser cet algorithme à SAT, nous avons été assez surpris par l'étendue des exemples sur lesquels le calcul de résolvant dépassait significativement en temps les meilleurs algorithmes systématiques mondiaux. Dans cette section, nous répertorions les exemples, pris dans la base de problème DIMACS [1], et nous avons choisis comme référence TABLEAU, Asat et C-SAT pour les temps CPU. Nous donnons dans le tableau 4.2 le résumé des résultats obtenus. Lorsque nous n'avons pas pu obtenir de résultat par manque de temps, la case est vide. De plus, le fichier pret150-75 a été prouvé grâce à l'approche hybride que l'on présente dans le chapitre suivant.

Dans le tableau ci-dessus, les premières lettres des noms de fichiers représentent une même classe de problèmes, générée selon la même méthode en faisant varier des paramètres. Les chiffres signifient différentes instances d'une même classe de problème. Tous les tests ont été effectués sur des Ultra-Sparc 2 avec 64 Mo de mémoire, et la version 7.1 de *Caml* pour les calculs de résolvants.

Cependant, il faut bien noter que d'une part, notre implémentation est en *byte-code* caml, ce qui signifie que le code est interprété pendant l'exécution, et que l'on peut avoir au moins un facteur de 10 à 20 par rapport à une implémentation optimisée en C. D'autre part, il se peut que les bons résultats de notre approche corresponde exactement à la classe de problèmes identifiée par Dechter et Rish dans [9]. Il faudrait prendre soin de vérifier la génération des exemples que nous venons de traiter efficacement pour voir dans quelle mesure ceux-ci correspondent à ce type de problème, adaptés au calcul de résolvant et inadaptés pour DP. Mais, par rapport à l'approche de Dechter et Rish, qui construisent une classe de problèmes montrant les avantages de leur méthode, nous avons puisé les exemples à traiter dans la base DIMACS, qui contient encore aujourd'hui des problèmes ouverts au problème fondamental SAT. Il serait étonnant que tous les exemples que nous traitons soient tous construits selon les mêmes principes, identifiés par Dechter et Rish. Encore une fois, cela reste à vérifier dans un travail à venir.
Enfin, les procédés de tests de DIMACS imposent de traiter toute une classe de problèmes, avec un temps de traitement de chaque instance en moins de 10000s. Nous ne donnons ici que des temps pour certaines instances de problèmes, car tous les tests n'ont pas pu être menés dans cette voie. Notre propos n'est donc pas vraiment de montrer que notre algorithme surpasse ceux cités sur des classes entières de problèmes (mis à part la classe de problème *Dubois*), mais plutôt qu'il existe des instances assez nombreuses de ces problèmes pour lesquelles notre algorithme semble particulièrement bien adapté. De plus, les méthodes systématiques permettent de traiter des exemples aléatoires de plus de 1000 clauses en quelques dixièmes de seconde. Notre approche est limitée à 200 clauses, dans des temps qui peuvent dépasser plusieurs heures sur certains exemples. Ces méthodes basées sur DPL surpassent généralement, d'un facteur significatif, notre algorithme de calcul de résolvant, sauf sur quelques classes ou instances de problèmes que l'on a pu trouver dans la base de DIMACS et qui ont été construites avec des structures étudiées pour tromper les heuristiques utilisées dans DP. Signalons tout de même que notre implémentation n'est en aucune manière optimisée pour les tests de satisfiabilité et on peut penser que nombre d'améliorations pourraient

1. Disponibles sur internet à l'adresse ftp.rutgers.edu

Exemple(s)	Nombre de clauses/variables	Temps ASat	Temps C-SAT	Temps Tableau	Temps calcul de résolvant	Satisfiable
dubois20	160c/60v	–	68s	72s	**1,5s**	Non
dubois21	168c/63v	–	143s	160s	**1,7s**	Non
dubois22	176c/66v	–	290s	380s	**1,8s**	Non
dubois23	184c/69v	–	584s	649s	**1,9s**	Non
dubois24	192c/72v	–	1190s	1320s	**2,2s**	Non
dubois25	200c/75v	–	2540s	3450s	**2,3s**	Non
dubois26	208c/78v	–	–	7334s	**2,5s**	Non
dubois27	216c/81v	–	–	>10000s	**2,7s**	Non
dubois28	224c/84v	–	–	>10000s	**2,8s**	Non
dubois29	232c/87v	–	–	>10000s	**3s**	Non
dubois30	240c/90v	–	–	>10000s	**6s**	Non
dubois50	400c/150v	>50min	>10000s	>10000s	**14s**	Non
ssa0432-003	1027c/435v	–	–	**4s**	40s	Non
ssa2670-130	3315c/1359v	>20min	–	>10000s	**15min**	Oui
ssa7552-038	3575c/1501v	–	–	>10000s	**11min**	Oui
ssa7552-158	3034c/1363v	–	–	>10h	**4min15s**	Oui
ssa7552-159	3028c/1363v	–	–	>10000s	**4min17s**	Oui
ssa6288-047	34238c/10410v	–	–	–	*14h*	Non
bf1355-075	6778c/2180v	>1h	–	**1min**	47min	Non
bf2670-001	3419c/1493v	–	>10000s	>1h30min	**10min15s**	Non
pret60-25	160c/60v	5min57s	–	2min65s	**7s**	Non
pret60-75	160c/60v	6min2s	–	52s	**9s**	Non
pret150-75*	400c/150v	–	–	>15000s	**3h**	Non
aim-50-1_6-no-1	76c/50v	**0.02s**	0.06s	0.07s	0.07s	Non

TAB. 4.2 - *Application du calcul de résolvant au problème SAT*

être envisagées dans le cas d'une restriction de l'algorithme à ce problème. Dans le chapitre suivant, nous présentons une approche hybride du calcul de résolvant, et on peut penser que cette approche, combinée avec des heuristiques adéquates pour les affectations des variables pendant les périodes hybrides pourrait donner encore de meilleurs résultats.

Les résultats obtenus sont bien sûr extrêmement encourageants et permettent de penser que le calcul de résolvant met à jour des caractéristiques absentes dans DP. Cependant, il faut bien noter ici que quelques séries de problèmes n'ont pas pu être traitées ni par notre méthode, ni par d'autres méthodes systématiques. Il y a par exemple toute une série d'exemples issus de l'expression en propositionnel de problèmes de planifications qui ont plus de 65000 clauses. Notre algorithme n'a pas pu traiter un seul de ces benchmarks (série g).

Chapitre 5

Une approche hybride pour le calcul de résolvant

On a vu dans la formalisation du calcul de résolvant que celui-ci est assez proche de la procédure de Davis et Putnam traditionnelle (DPL). Or, la puissance de DPL vient du fait qu'on ne procède qu'à des simplifications (propagation d'affectations entraînant l'élimination des clauses vérifiées par le littéral), alors que dans notre approche, aucune simplification de ce type a lieu. Ce chapitre tente de faire un lien entre les deux procédures et adapte DPL au problème de calcul de résolvant.
Mais, si l'on utilise uniquement l'adaptation de DPL au calcul de résolvant, on se trouve avec un algorithme qui doit explorer toutes les branches de l'arbre de Shannon de la formule traitée. Or, cet arbre a clairement un nombre exponentiel de branches. On ne peut donc pas raisonnablement penser appliquer uniquement DPL au calcul de résolvant sur plus d'une vingtaine de variables à éliminer.
Nous introduisons donc une sorte de Méta-Heuristique, qui se propose de changer de méthode de calcul pendant ce même calcul, permettant de passer du calcul de résolvant classique (vu aux chapitres précédents) au calcul de résolvant par DPL. On a vu en redécouvrant l'algorithme initial de Davis et Putnam que chaque algorithme détenait ses propres caractéristiques, plus ou moins adaptées à certaines classes de problèmes. Si Dechter et Rish dans [10] ont pu montrés le gain de performance d'une méthode hybride pour SAT, on tente l'approche inverse, en adaptant les algorithmes traditionnels pour SAT au problème particulier du calcul de résolvant.

Après la définition de l'approche hybride que nous proposons, la suite du chapitre traite des heuristiques nécessaires à la gestion par le programme hybride du basculement d'une méthode à l'autre en cours de calcul.

5.1 Motivations

Le nombre de clauses gérées par le programme a tendance à croître exponentiellement avec le nombre de clauses initiales. Mais, on peut voir sur les exemples aléatoires que pour éliminer toutes les variables, le programme passe par un maximum de clauses pour voir ensuite ce nombre redescendre, sans pour autant voir la longueur des clauses augmenter de manière significative. Si on étend ces observations aux exemples concrets sur lesquels on a pu travailler, il est naturel de tenter de trouver le moyen de passer ces pics de nombre de clauses. Une des alternatives au calcul de résolvant présenté dans les chapitres précédents est l'adaptation de DPL au problème présenté, DPL ne procédant qu'à des simplifications.

5.1.1 Observations de bosses de difficultés

Le passage par un maximum de nombre de clauses semble être une constante dans les exemples aléatoires sur lesquels nous avons mené nos expérimentations. Ce maximum a été clairement identifié au cours de la section 3.5.2. Il est certes plus difficile de généraliser cette observation aux problèmes réels, mais on a pu observer plusieurs séries de problèmes difficilement traitables avec l'ancienne méthode et plus

adaptées à la méthode hybride que nous présentons.
À ces maxima de nombre de clauses, correspond un maximum de temps nécessaire par coupure. Si on analyse plus en détail ces bosses, on peut estimer raisonnablement trouver le nombre de coupures *difficiles* avant de retomber dans des temps par coupure raisonnables (*par exemple, les 8 prochaînes coupures sont les plus coûteuses*). On le voit ici, le problème de cette approche hybride est justement de trouver a priori le *bon* moment de passage à DPL, et la *bonne* durée de ce passage.

5.1.2 Arbre de décomposition de Shannon

On appelle *arbre de décomposition de Shannon* l'arbre binaire ordonné dont chaque noeud correspond à la décomposition de Shannon d'une seule variable. Les feuilles de cet arbre sont soit 1, soit 0. Par exemple, un arbre de décomposition de Shannon à de profondeur 2 est donné figure 5.1. On voit que si on avait fini le développement de l'arbre, chaque feuille aurait été égale à 0 ou 1. Nous avons choisi de ne pas le développer jusqu'au bout pour bien montrer que le résolvant de la formule sur les deux variables a et b est en fait la disjonction des feuilles de l'arbre schématisé.

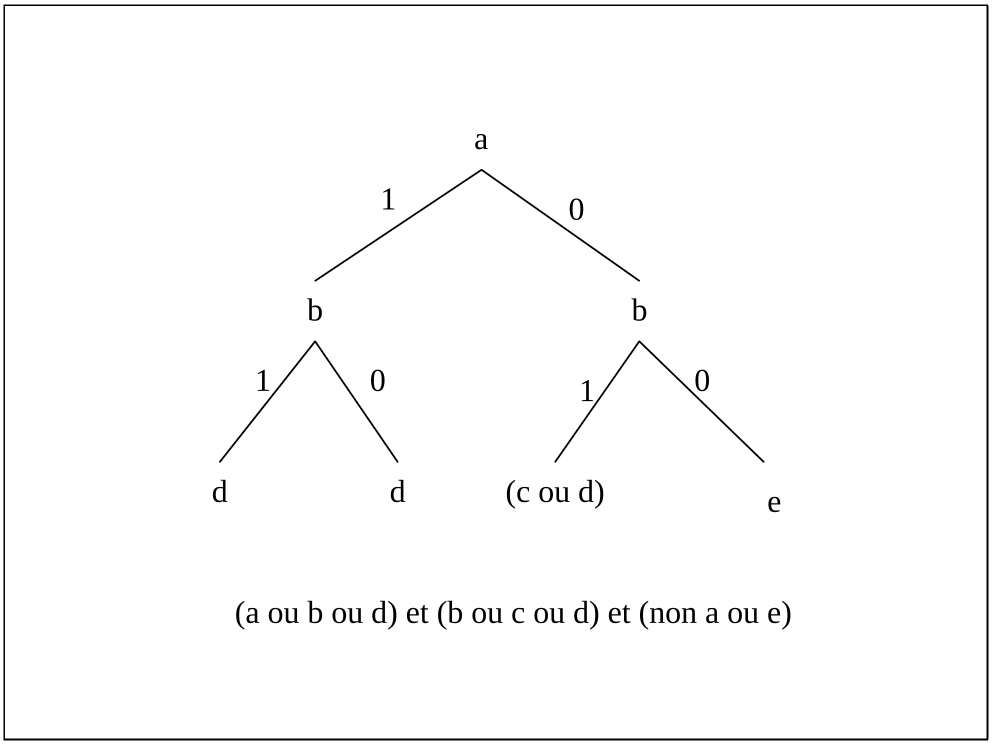

FIG. 5.1 - *Arbre de décomposition de Shannon*

5.1.3 Une façon de limiter l'explosion du nombre de clause

Même avec une structure de données comme le Trie, le nombre de clauses manipulées reste clairement une limite à l'approche que nous avons présentée. Une des possibilités pour limiter ce nombre prohibitif est de faire des affectations de variable qui vont permettre d'éliminer toutes les clauses contenant le littéral vérifié par l'affectation. Cependant, à la différence de DPL qui doit intuitivement trouver un chemin dans l'arbre de décomposition de Shannon de la formule (introduit en 2.2), le calcul de résolvant implique de prendre en considération tous les chemins de l'arbre de décomposition de Shannon. Or, cet arbre contient, pour pratiquement toutes les formules, de l'ordre de $O(2^n)$ noeuds. On peut d'ailleurs voir ici une application directe, du fait que le problème n'est pas uniquement NP-Complet, mais qu'il est aussi CoNP-Complet (on peut difficilement espérer trouver un algorithme rapide, par rapport à DPL, qui

trouve le résolvant d'une formule).
Il faut donc introduire une profondeur maximale sur laquelle on peut raisonnablement lancer l'approche hybride. Cette profondeur doit clairement être inférieure à une borne constante, par exemple 32, puisque on ne peut pas espérer explorer quatre mille milliards de feuilles dans un temps raisonnable. Il faut bien préciser qu'à notre sens il n'existe pas vraiment d'autres moyens que cette approche pour limiter le nombre de clauses pendant le calcul, même si elle semble d'un coût beaucoup trop prohibitif en temps. Cependant, et on va le voir dans l'amélioration significative des résultats, cette approche, même bornée très fortement par des petits nombres, peut s'avérer très rentable. Voyons d'abord une formalisation plus précise de notre approche hybride.

5.2 Formalisation de l'approche hybride

Comme dans le cadre de la méthode classique, on donne ici une formalisation de l'approche hybride. Cette formalisation se résume principalement à la démonstration de l'équivalence des méthodes proposées sur le résolvant produit à la fin du calcul.

5.2.1 Équivalence des deux méthodes

On a vu que le but de l'approche hybride est de couper le calcul en plusieurs sous-calculs. Pour cela, il faut établir la proposition suivante, qui nous garantit l'équivalence de toutes les méthodes proposées :

Proposition 5 *Le résolvant de la formule $A \vee B$, où A et B sont des formules propositionnelles, est équivalent à la disjonction des deux résolvants respectifs des formules A et B.*
Soit,

$$Res_I(\ A \ \vee \ B\ ) = Res_I(A) \ \vee \ Res_I(B)$$

où $Res_I(f)$ représente le résolvant de f sur les variables I.

En démontrant cette proposition, on prouve que toute formule qui peut être écrite par une disjonction de deux formules (chacune étant plus simple que la formule initiale) a un résolvant qui peut se calculer par la disjonction des résolvants des deux formules. Ainsi, on peut utiliser la décomposition de Shannon pour faire des simplifications de formules.
Montrons par récurrence que cette proposition est vérifiée pour $I = \{v\} \cup J$.

Supposons que le proposition soit vraie pour J, donc

$$Res_J(\ A \ \vee \ B\ ) = Res_J(A) \ \vee \ Res_J(B)$$

On a montré dans la section 2.2 que le résolvant sur la variable v de f pouvait s'écrire

$$(f_{v\leftarrow 1}) \vee (f_{v\leftarrow 0})$$

Donc $Res_v(A \ \vee \ B)$ peut s'écrire, en substituant f par $A \ \vee \ B$

$$(A_{v\leftarrow 1} \vee B_{v\leftarrow 1}) \vee (A_{v\leftarrow 0} \vee B_{v\leftarrow 0})$$

ou :

$$(A_{v\leftarrow 1} \vee A_{v\leftarrow 0}) \vee (B_{v\leftarrow 1} \vee B_{v\leftarrow 0})$$

Ce qui s'écrit en fonction du résolvant des formules de A et B :

$$Res_v(A) \vee Res_v(B)$$

On a donc prouvé que $Res_v(A \vee B) = Res_v(A) \vee Res_v(B)$

La propriété est donc vérifiée pour I réduit à un singleton.

Enfin, comme $Res_I(f) = Res_J(Res_v(f))$, il vient, en remplacant f et en appliquant l'égalité que l'on vient de prouver :

$$Res_I(A \vee B) = Res_J(Res_v(A) \vee Res_v(B))$$

Par application de l'hypothèse de récurrence, il vient

$$Res_I(A \vee B) = Res_J(Res_v(A)) \vee Res_J(Res_v(B))$$

ce qui est bien

$$Res_I(A \vee B) = Res_I(A) \vee Res_I(B)$$

On a donc montré que la propriété était vérifiée pour un ensemble de variables réduit à un singleton. On a montré d'autre part que si la propriété était vraie pour l'ensemble des variables J, alors elle était aussi vraie pour $I = J \cup \{v\}$. Donc, par récurrence sur l'ensemble des variables, la propriété est vraie sur tout ensemble de variables.

Une fois cette équivalence définie, on peut appliquer à tout moment la décomposition de Shannon sur une variable et découper ainsi le calcul de résolvant d'une formule en deux calculs de résolvants plus simples (chacun correspondant à une affectation de la variable entrant dans la décomposition de Shannon).

5.2.2 Deux approches hybrides possibles

On peut imaginer deux sortes d'approches hybrides. L'une peut calculer les disjonctions des résolvants obtenus sur les variables simplifiées puis lancer le calcul de résolvant sur la nouvelle formule obtenue après le passage en DPL. L'autre approche peut lancer des calculs de résolvant sur chaque feuille de l'arbre de décomposition de Shannon puis calculer la disjonction de tous les résolvants à la fin des calculs des résolvants de chaque branche. On peut schématiser les deux approches hybrides possibles par la figure 5.2.

En fait, la première approche hybride, que nous appellerons *hybride-bornée*, peut être efficacement utilisée lors du passage de bosse. Son principe consiste à distribuer les clauses issues de toutes les simplifications avant de relancer le calcul sur le résolvant obtenu. Si on est pratiquement sûr de pouvoir retrouver un résolvant simple à la fin de la période hybride, on peut penser y gagner à tout distribuer avant de ne relancer le calcul que sur une branche.
La seconde approche, que nous appellerons *hybride-finale*, divise un calcul de résolvant en 2^n calculs de résolvants, plus simples. Ceci peut être utilisé dans le cas de résolvants plus difficiles à obtenir, ou dans le cas de passage de bosse trop long pour espérer retomber avec une profondeur raisonnable sur un calcul de résolvant plus simple. Cette dernière approche est la seule garantissant uniquement une simplification des formules, au détriment d'un nombre exponentiel de formules (chacune plus simple) à traiter.

5.3 Des heuristiques de base pour l'approche hybride

L'introduction de la possibilité de passer à *hybride-bornée* ou *hybride-finale* implique l'introduction d'heuristiques permettant de passer d'une méthode à l'autre. Les différentes heuristiques que nous avons pu identifier sont présentées dans cette section. Après un point sur les heuristiques utilisées dans DPL, nous présentons celles que nous avons pu implémenter et tester.

Signalons toutefois que nous n'avons pas trouvé actuellement d'heuristique permettant de faire un choix entre *hybride-bornée* et *hybride-finale*. Ce choix a été laissé au programmeur, celui-ci choisissant l'une ou l'autre des approches hybrides selon le type de formule sur lequel il lancait le calcul de résolvant.

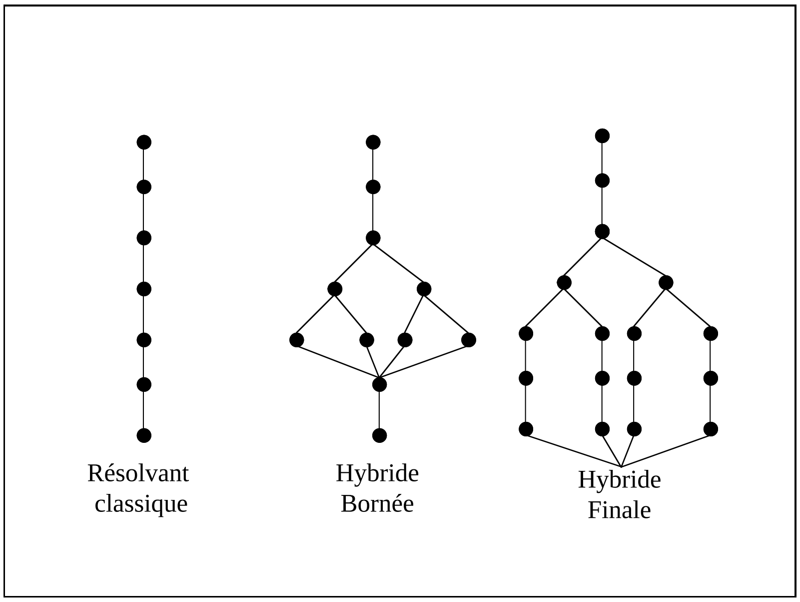

FIG. 5.2 - *Les 3 approches du calcul de résolvant*

5.3.1 Heuristiques traditionnelles pour DPL

Les dernières avancées dans la taille des problèmes 3-SAT aléatoires traitables sont en partie dues aux heuristiques mises en place dans le choix du chemin dans l'arbre de décomposition de Shannon. L'heuristique traditionnellement utilisée est celle consistant à choisir la variable la plus fréquente dans les clauses raccourcies (clauses affectées par des affectations antérieures) et à lui attribuer la valeur du littéral satisfaisant le maximum de clauses. Cependant, on ne peut pas espérer appliquer directement ce genre d'heuristiques au problème de calcul de résolvant. En effet, notre problème consiste à explorer toutes les feuilles de l'arbre de Shannon sur les variables de l'approche hybride, et non pas à trouver rapidement la branche qui mène à la preuve de la satifiabilité de la formule.
Il reste malgré tout des études à mener dans l'adaptation des meilleures heuristiques de DP, en vue de prendre en compte une augmentation des chances d'obtenir des clauses unitaires, des variables unitaires, ou des contradictions dans toutes les formules obtenues aux feuilles de la décomposition de Shannon.

5.3.2 Tenter ou réfléchir : quelle profondeur pour chacun d'eux ?

Par analogie à [10], on peut voir la différence entre les deux approches se mêlant dans la procédure hybride comme :

- L'une prenant la peine de réfléchir (on prend les deux possibilités en compte et on regarde le résultats), ce qui correspond à l'approche classique du calcul de résolvant.
- L'autre tentant des affectations des variables à éliminer, regardant les effets produits dans la formule et distribuant tous les résultats obtenus. Cette approche est celle de DPL.

Le problème de passer d'une méthode à l'autre se résume principalement au choix de la profondeur optimale de l'une ou de l'autre méthode.

Une profondeur dynamique pour le résolvant

En ce qui concerne le calcul de résolvant, la profondeur peut être dynamique, c'est-à-dire que le choix de continuer à utiliser la méthode classique peut être remis en cause après chaque coupure. On ne s'engage que sur une coupure à chaque début de calcul. Il reste à trouver le bon moment pour lancer la partie de simplification par affectation de variables. Cette question est traitée en 5.3.3.

Trouver a priori une bonne profondeur pour DPL

La bonne profondeur pour DPL est facilement bornable par un maximum (par exemple 30). Pourtant, il faut trouver une bonne profondeur a priori pour DPL, faisant la balance entre un nombre exponentiel de branches et une simplification maximale de la formule initiale. En effet, lorsque l'on choisit de lancer une partie DPL du calcul hybride, on doit s'y tenir jusqu'à la fin du calcul, d'où l'importance du choix de la profondeur optimale. Dans le cas de problèmes aléatoires, cette profondeur peut être approximée grâce à l'expérimentation que nous avons faite dans les chapitres précédents. Ainsi, on a une bonne idée du nombre de clauses correspondant à la nième coupure. Nous avons tout de même mené une petite expérimentation, sur une classe de formules aléatoires de 180 clauses *difficiles*, pour comparer les différents résultats en fonction de la profondeur choisie. Ces résultats sont analysés dans la section 5.4.3.

5.3.3 Tenter ou réfléchir : quand basculer de l'un à l'autre ?

Sur des classes de problèmes uniformes comme les problèmes aléatoires, on peut appliquer une borne fixe de basculement dans l'approche hybride. Cette borne peut être trouvée toujours grâce aux expérimentations menées sur les problèmes aléatoires.
Pour les problèmes concrets, on peut trouver une heuristique simple, permettant de trouver un moment de basculement nécessaire :

- Dans le cas où le nombre de clauses produites par une coupure est impraticable (par exemple, dans *Parity32*, on se trouve avec des coupures sur des variables les moins fréquentes apparaissant 7580 fois sous forme positive et autant sous forme négative : la coupure produirait plus de 57 Millions de clauses). Dans un tel cas, la seule solution est de simplifier la formule, donc de lancer une approche hybride.

Nous n'avons implémenté que cette heuristique dans le programme. De plus, nous avons ajouté la liberté au programmeur surveillant l'évolution du programme de lancer l'approche hybride à n'importe quel moment du calcul. Ainsi, il est libre de pousser le programme à passer dans un mode DPL en surveillant lui-même l'évolution des coupures et du nombre de clauses.

5.3.4 À profondeur fixée : quelles variables choisir ?

Le dernier problème à résoudre est celui concernant le choix des variables dont on va fixer la valeur pendant la période DPL du calcul hybride. Ce choix doit être guidé par un souci de simplification maximale sur toutes les formules des feuilles de l'arbre de décomposition de Shannon. Pour cela, le choix doit s'orienter vers les variables les plus fréquentes, à la fois sous forme positive et sous forme négative. L'heuristique la plus prometteuse serait celle qui pourrait choisir les variables touchant le plus grand nombre de clauses, *dichotomisant* le plus la formule initiale. Nous avons choisi, dans un souci de respect du temps imparti pour l'implémentation, de ne tester qu'une heuristique, assez simple, basée sur le principe suivant :

$$\textit{Choisir les } n \textit{ variables } v \textit{ telles que } Card(Cpos(v))^2 \; + \; Card(Cneg(v))^2 \textit{ soit maximal}$$

Cette heuristique permet de préférer une variable apparaissant 20 fois sous forme négative et positive plutôt qu'une variable apparaissant 50 fois sous une forme et 1 fois sous une autre. Ceci permet d'obtenir au maximum des formules toutes simplifiés dans les feuilles de l'arbre de Shannon.
On voit en énonçant cette heuristique qu'il peut y avoir des formules dont la dichotomisation maximale pourrait avoir lieu avec des séries de variables ne répondant pas au critère heuristique que l'on vient d'énoncer. Encore une fois, l'identification de telles variables pourrait être faite au cours d'un travail ultérieur. L'heuristique implémentée ici donne cependant une bonne idée des performances de la méthode hybride, comme on peut le voir dans la section suivante.

5.4 Résultats expérimentaux

5.4.1 Classes de problèmes pour cette approche

Lors des expérimentations, le coût de la distribution des résolvants en fin de période DPL s'est avéré très élevé. Les problèmes traitables par cette approche sont tous ceux qui ne contiennent que peu de clauses dans le résolvant final, permettant une distribution des résolvants rapide, ou ceux qui contiennent certaines variables très fréquentes et dont l'affectation provoque beaucoup de simplifications dans la formule en cours de traitement.
Les problèmes concrets que l'on a pu traiter par cette approche sont des problèmes où l'on cherche à éliminer toutes les variables (on résout en fait le problème SAT). Les résolvants sont alors tous soit vides, soit égales à la clause vide et leur distribution finale a un coût pratiquement nul. Nous n'avons pas pu trouver d'exemple correspondant à des problèmes concrets contenant un faible nombre de clauses dans le résolvant et dont l'approche classique de calcul de résolvant n'arrivait pas à bout.
Pour les problèmes aléatoires, nous avons calculé les résolvants sur 3 ou 4 variables restantes.

5.4.2 Problèmes concrets

L'approche hybride n'a pas permis de résoudre de nouveaux problèmes modélisant des arbres de défaillance de systèmes physiques. En effet, la classe de problèmes sur laquelle le programme butait était en fait celle qui contenant plusieurs dizaines de milliers de clauses dans le résolvant, voire plusieurs centaines de milliers. Même si le programme arrive à manipuler jusqu'à plus de 60000 clauses avant un dépassement des capacités mémoire des machines sur lesquelless nous effectuions les tests, les coupures menant à un tel nombre de clauses étaient d'une manière pratique irréalisables. De plus, l'approche hybride projette en fait le problème des coupures sur la distribution des résolvants. Or, dans le cas où ces derniers contiennent un nombre impratiquable de clauses, l'approche hybride n'apporte rien.
Mais, nous avons trouvé des exemples de problèmes structurés dans la littérature sur SAT. Les résolvants sont alors soit vides, soit égales à la clause vide. Dans ce cas, la distribution finale ne coûte rien et on a pu traiter certains problèmes impossibles avec l'approche classique. Ces exemples ont été lancés avec un passage en DPL automatique en cas de coupure produisant plus de 60000 clauses. La profondeur de DPL est fixée à 5, mais la période DPL peut être lancée plusieurs fois de suite.
On donne ici quelques résultats sur des exemples intraitables avec l'approche classique.

- ii32b1, 1374c/228v, 19801s
- Parity16, 3310c/1015v, 78025s
- Hole10, 561c/110v, 131000s

Nous n'avons pas pu exécuter le programme sur beaucoup plus d'exemples, en raison du temps important pris pour la résolution d'un seul exemple. On pouvait s'attendre à de tels résultats en raison de la croissance exponentielle du temps par rapport à la simplification effectuée dans les formules par les périodes DPL.

5.4.3 Problèmes aléatoires

On a pu s'apercevoir que sur les exemples aléatoires, les simplifications effectuées par la période DPL de l'approche hybride n'entraînent pas de changement suffisant dans la formule. Ainsi, après une simplification, on retrouve en quelques coupures le même ordre de grandeur dans les fréquences minimales des variables. Ce résultat tend donc dans le même sens que celui de l'approche hybride pour SAT de Dechter et Rish [10], dans lequel les résultats expérimentaux de leur algorithme DPL avec des périodes d'élimination de variables par calcul de résolvant obtenaient approximativement les mêmes résultats que l'algorithme DPL seul.

Malgré ces observations, l'approche *hybride-finale* a tout de même permis d'augmenter la taille des formules traitable, du moment que ces dernières n'avaient pas un résolvant trop grand. Par contre, *hybride-bornée* n'a pas permis de taiter des formules systématiquement plus grandes qu'avec l'approche classique. Nous avons donc mené une petite série d'expérimentations sur les formules aléatoires en calculant le résolvant, par *hybride-finale*, sur 4 variables de formules de 250 clauses dans la zone *difficile* de 3-SAT. Le calcul de résolvant est à chaque fois arrivé à bout, en procèdant généralement à deux séries de simplifications par 5 variables. La distribution finale correspond donc dans ce cas à 1024 résolvants issus de chaque branche de développement simplifiée. Ces résultats ont été obtenus en moyenne en 21200s sur 8 exemples de 250 clauses et 58 variables sur un ordinateur personnel et une moyenne de 25600s sur des exemples de 300 clauses, 70 variables sur un Ultra Sparc 2. Ceci peut tout de même être vu comme un bon point de l'algorithme, puisque même avec des résolvants sur 4 variables, l'approche précédente était limitée à 200 clauses.

Nous sommes consciens que l'approche hybride n'est pas forçément la solution au problème de calcul de résolvant, surtout quand les résolvant peuvent contenir plusieurs centaines de milliers de clauses. Elle reste cependant une bonne technique de simplification des formules, et permet ainsi, dans le cas de résolvant de taille *raisonnable*, de projeter la consommation mémoire du programme vers une consommation de temps.

5.5 Notes sur l'implémentation des algorithmes

L'implémentation des algorithmes que nous avons présenté, l'utilisation de structures de données adaptées, la mise au point de plusieurs heuristiques et l'exécution de plusieurs dizaines de milliers de fois du programme a nécessité un gros travail d'implémentation, qui fait partie intégrante du travail de ce stage de DEA. Mais, nous n'avons pas voulu alourdir ce rapport par les 4000 lignes de code *Caml* programmées, et nous n'avons pas mentionné les problèmes rencontrés pour obtenir une automatisation des tests sur des classes d'exemples aléatoires. Ceci a bien entendu posé nombre de problèmes d'implémentation qui ont naturellement ralenti le travail mené, limité dans le temps. Cependant, les problèmes d'implémentation restent des problèmes classiques et n'entrent pas vraiment dans le cadre de ce rapport. Il fallait juste le mentionner ici.

Chapitre 6

Vers une méthode de représentation implicite du résolvant

Le travail présenté dans ce chapitre se détache nettement du reste du rapport. Il a pour but de montrer qu'il peut exister d'autres techniques fondamentalement différentes pour le calcul de résolvant. Notre travail reste cependant très prospectif, effectué en fin de stage, et rien ne nous permet encore de dire avec certitude s'il permettra une réelle amélioration des résultats même si l'approche que nous présentons devrait théoriquement avoir une complexité indépendante du nombre de clauses manipulées. Nous sommes consciens que la lecture de ce chapitre implique une bonne connaissance des ROBDD et des méthodes de calculs d'impliquants premiers proposées par Madre et Coudert en 1994. Même si nous réintroduisons ces concepts dans ce chapitre, certains résultats sont exposés sans les justifications qui pourraient permettre de comprendre pleinement les nouveaux algorithmes que nous présentons, mais ces justifications feraient largement déborder ce rapport de DEA et nous tenions pleinement à présenter la voie que nous avons ouverte.

6.1 Principes

Les chapitres précédents ont bien montré que la limite au traitement de l'algorithme de calcul de résolvant vient essentiellement du nombre de clauses à traiter lors de la coupure et lors des tests de subsomptions. Or, en comparant les résultats obtenus avec ceux des programmes calculant l'ensemble des impliquants premiers, dont un lien est donné en 2.6.1, les performances de la méthode que nous présentons ont été considérablement dépassées par ces algorithmes spécialisés dans le cas de modélisation de systèmes réels (l'exemple *Baobab* n'a pas pu être traité par le programme alors qu'il est complètement analysé par *Aralia* ([21]) en moins de $0.1s$).

On peut être certain que ces différences ne viennent ni du choix du langage ni d'optimisations dans l'implémentation des algorithmes. La méthode employée a été présentée pour la première fois en 1994 par J.-C. Madre et O. Coudert ([13]), et a ouvert de nouveaux champs d'horizons dans les exemples pouvant être traités.

Dans ce chapitre, nous allons voir comment on peut espérer adapter ces méthodes, ou tout du moins leur idée fondamentale, au calcul de résolvant, sans pour autant retrouver un des inconvénients majeurs (voir 6.3.3) de celles-ci. Après un descriptif de ces méthodes et de leurs structures de données grandement liées à leur puissance, nous verrons comment définir la distribution des clauses lors des coupures et comment faire de nouveaux tests de subsomptions en se basant exclusivement sur ces méthodes. Pour finir, nous définissons plusieurs nouveaux algorithmes possibles pour le calcul de résolvant lui-même. Ces algorithmes ne sont cependant pas implémentés, par manque de temps, ce qui empêche toute comparaison empirique entre eux. On se contente donc, autant que possible, de dresser un panorama exhaustif de ce qu'il serait possible de faire, en utilisant les méthodes de représentation implicite, dans une suite de ce travail.

6.1.1 Motivation : changement d'échelle avec les méthodes implicites

Lors de la présentation de leurs résultats dans [13], J.-C. Madre et O. Coudert ont noté un changement d'échelle dans les problèmes traitables d'analyses d'arbres de défaillance. Les techniques utilisées jusqu'alors possédaient une complexité directement liée au nombre d'impliquants premiers manipulés. Les limites de ces techniques s'arrêtaient aux fonctions de plus de 20000 impliquants premiers (*baobab* en contient quelques 46188). En ne manipulant plus directement les ensembles d'implicants premiers, mais les fonctions caractéristiques de ces mêmes ensembles, Madre et Coudert ont permis de gagner un facteur temps de 1000 par rapport aux techniques traditionnelles, de résoudre 25 problèmes encore ouverts en 1994 et de traiter des fonctions avec plus de 10^{20} impliquants premiers. Le codage des fonctions caractéristiques utilise des ROBDD, définies en 6.2, qui permettent d'avoir une complexité ne dépendant absolument pas du nombre d'éléments que celles-ci représentent. On peut donc espérer obtenir des algorithmes ne subissant pas la bosse de difficulté que l'on a pu observer, liée au nombre de clauses manipulées.

MétaProduit et représentation d'un ensemble de clauses (MétaSomme)

Désireux de manipuler des ensembles d'impliquants premiers (formules mises sous FND), J.-C. Madre et O. Coudert ont défini une représentation implicite d'ensembles de produits, en codant la fonction caractéristique de ces ensembles.

Dans [13], les auteurs définissent ce qui suit :
Un *littéral* est une variable propositionnelle x_k ou sa négation, qu'ils notent aussi $\overline{x_k}$. L'ensemble P_n des *produits* construits à partir de l'ensemble des variables propositionnelles V_n est l'ensemble $\{x_1, \overline{x_k}, \epsilon\} \times \ldots \times \{x_n, \overline{x_k}, \epsilon\}$, où ϵ est la chaîne vide. Un *produit* est un élément de P_n , et est interprété comme la conjonction logique des littéraux le composant.

Dans le but de manipuler directement les ensembles de produits par leurs fonctions caractéristiques, les auteurs ont défini le *MétaProduit* $MP(\pi)$ codant un produit π comme la conjonction, sur toutes les variables x présentes dans la formule, des $mp(\pi, x)$, où $mp(\pi, x)$ est défini par :

$$mp(\pi, x) = \begin{cases} (o_x \wedge s_x) & si\ x \in \pi \\ (o_x \wedge \overline{s_x}) & si\ \overline{x} \in \pi \\ \overline{o_x} & sinon \end{cases}$$

Les variables o_i représentent l'occurence de la variable x_i et s_i représente le signe du littéral. Un ensemble de produit (formule sous FND) est codé par la disjonction de tous ses MétaProduits. Par exemple, la formule

$$(a \wedge \overline{b}) \vee (b \wedge c \wedge \overline{d}) \vee (\overline{a} \wedge c)$$

est codée par le MétaProduit suivant :

$$(o_a \wedge s_a \wedge o_b \wedge \overline{s_b} \wedge \overline{o_c} \wedge \overline{o_d}) \vee (\overline{o_a} \wedge o_b \wedge s_b \wedge o_c \wedge s_c \wedge o_d \wedge \overline{s_d}) \vee (o_a \wedge \overline{s_a} \wedge \overline{o_b} \wedge o_c \wedge s_c \wedge \overline{o_d})$$

Cette dernière formule, qui paraît de prime abord beaucoup plus complexe à représenter que la formule initiale a l'avantage de pouvoir être vue comme la fonction caractéristique de l'ensemble des produits codés. Elle est définie sur $([o_1 \ldots o_n] \times [s_1 \ldots s_n])$ à valeurs dans *Bool*. Or, une telle représentation permet de définir des calculs ensemblistes efficaces, basés sur des opérations logiques entre les fonctions caractéristiques (voir 6.1.2). Ce que l'on semble perdre en taille de données, on le gagne intuitivement en coût d'opérations. De plus, en représentant les fonctions caractéristiques, qui ne sont autres que des formules booléennes, par des ROBDD, on permet de représenter ces formules de manière ultra-compacte. Une vision d'ensemble des ROBDD est donnée dans la section suivante.

MétaSomme

Pour rester dans la lignée de Madre et Coudert, et pour décrire formellement un ensemble de clauses par sa fonction caractéristique que nous coderons plus tard par un ROBDD, il suffit de définir une *MétaSomme* comme le dual du MétaProduit. L'ensemble S_n des *sommes*, construit à partir de l'ensemble des variables propositionnelles V_n est l'ensemble $\{x_1, \overline{x_k}, \epsilon\} \times \ldots \times \{x_n, \overline{x_k}, \epsilon\}$, où ϵ est la chaîne vide. Une *somme* est un élément de S_n , et est interprétée comme la disjonction logique des littéraux le composant. On peut donc définir la MétaSomme $MS(\sigma)$ codant une somme σ comme la disjonction, sur toutes les variables x présentes dans la formule, des $ms(\sigma, x)$, où $ms(\sigma, x)$ est défini par :

$$ms(\sigma, x) = \begin{cases} (o_x \wedge s_x) & si\ x \in \sigma \\ (o_x \wedge \overline{s_x}) & si\ \overline{x} \in \sigma \\ \overline{o_x} & sinon \end{cases}$$

Un ensemble de sommes (formule sous FNC) est alors la conjonction de toutes ses MétaSommes. En prenant l'exemple dual du précédent, on peut voir que la MétaSomme

$$(o_a \wedge s_a \wedge o_b \wedge \overline{s_b} \wedge \overline{o_c} \wedge \overline{o_d}) \vee (\overline{o_a} \wedge o_b \wedge s_b \wedge o_c \wedge s_c \wedge o_d \wedge \overline{s_d}) \vee (o_a \wedge \overline{s_a} \wedge \overline{o_b} \wedge o_c \wedge s_c \wedge \overline{o_d})$$

code la formule suivante :

$$(a \vee \overline{b}) \wedge (b \vee c \vee \overline{d}) \wedge (\overline{a} \vee c)$$

Encore une fois, ce codage permet de manipuler des ensembles de clauses uniquement par opérations logiques sur leurs fonctions caractéristiques.

6.1.2 Manipulations d'ensembles sur les fonctions caractéristiques

Pour voir en quoi la manipulation de MétaSomme permet de gagner en complexité par rapport à la manipulation explicite d'ensembles de clauses, il suffit de montrer que les opérateurs usuels de manipulation d'ensembles se réécrivent en opérations logiques sur les MétaSomme. Ainsi,

- L'ensemble vide de clause est codé par la constante booléenne 0.
- L'ensemble de toutes les clauses est codé par 1.
- La clause vide est codée par $(\neg O_1 \wedge \cdots \wedge \neg O_n)$
- $\Sigma_1 \cup \Sigma_2$ est codé par la fonction $MS(\Sigma_1) \vee MS(\Sigma_2)$.
- $\Sigma_1 \cap \Sigma_2$ est codé par la fonction $MS(\Sigma_1) \wedge MS(\Sigma_2)$
- Le complément $\overline{\Sigma}$ de Σ est codé par la fonction $\neg MS(\Sigma)$.

Or les opérations logiques sur les MétaSomme ont un coût uniquement lié à la taille de la représentation de la formule booléenne représentée par la MétaSomme.

6.2 Les ROBDD, une représentation compactée des fonctions booléennes

En dehors de la mise sous forme normale conjonctive (FNC) ou disjonctive (FND) d'une formule propositionnelle, il existe d'autres formes normales, ayant toutes certaines propriétés désirables. La forme normale de Shannon (FNS) est basée uniquement sur les constantes 0 et 1 ainsi que sur l'opérateur *ite* (pour *If-Then-Else*), qui est défini comme suit :
Soient f, g et h trois fonctions booléennes,

$$ite(f, g, h) = (f \wedge g) \vee (\neg f \wedge h)$$

Une des propriétés remarquables de l'opérateur *ite* est l'orthogonalité vis-à-vis des connecteurs standard de la logique ($\vee$, $\wedge$, $\neg$). Un opérateur Op binaire est dit *orthogonal* avec *ite* si on a :

$$Op(f,g) \;=\; ite(x, Op(f_{x\leftarrow 1}, g_{x\leftarrow 1}), Op(f_{x\leftarrow 0}, g_{x\leftarrow 0}))$$

C'est cette propriété qui garantit clairement que toute formule basée sur les connecteurs usuels peut être mise sous FNS, par simple application récursive de l'orthogonalité.

6.2.1 Reduced Ordered Binary Decision Diagram

Une approche progressive des ROBDD et des structures de données apparentées peut être trouvée dans [1], support de cours de H.R. Andersen. Voir aussi [5, 3, 19] pour un survol complet sur les BDD. Les définitions suivantes sont inspirées de [20].
Un *Binary Decision Diagram* sur n variables $x_1, \ldots, x_n$ est un graphe acyclique dirigé α (DAG, pour *directed acyclic graph*) vérifiant les conditions *(i)* et *(ii)* :

(i). Les feuilles du graphe α sont étiquetées soit par 0, soit par 1.

(ii). Chaque noeud de α, qui n'est pas une feuille, est étiqueté par une variable x_i et a deux arcs sortants étiquetés par 0 et 1 respectivement.

La figure 6.1 schématise les différentes définitions des structures de données liées aux BDD. On écrit dans la suite $\Delta(x, \alpha_1, \alpha_0)$ pour le BDD dont la racine est étiquetée par la variable x et dont les deux arcs sortant-1 et sortant-0 pointent respectivement vers les deux BDD α_1 et α_0.
On appelle *Ordered Binary Decision Diagram* (OBDD) le BDD muni d'un ordre total $<$ sur les variables $x_1, \ldots, x_n$ qui respecte la condition *(iii)* :

(iii). Pour tout noeud interne $\Delta(x_i, \alpha_1, \alpha_0)$ du graphe α, on a :

- *α_1 est soit une feuille, soit un noeud étiqueté par une variable x_j tel que $x_i < x_j$.*
- *α_0 est soit une feuille, soit un noeud étiqueté par une variable x_k tel que $x_i < x_k$.*

Pour limiter l'utilisation de le mémoire, on peut ajouter la condition *(iv)*, ce qui donne, avec les règles précédentes, la définition d'un *Quasi-Reduced Ordered Binary Decision Diagram* (QOBDD).

(iv). le graphe α ne contient pas deux sous-graphes isomorphiques.

Enfin, les *Reduced Ordered Binary Decision Diagram* (ROBDD) sont les QOBDD satisfaisant, en plus, la propriété *(v)* :

(v). le graphe α ne contient aucun noeud de la forme $\Delta(x, \beta, \beta)$.

Si dans la définition, les ROBDD ne sont qu'une structure de données tentant de mettre en commun le maximum de sous-arbres identiques, ils sont une bonne structure de données pour représenter une formule sous la forme normale de Shannon. Dans le cas d'une telle utilisation des ROBDD, on a le théorème suivant :

THÉORÈME 4 (Canonicité des ROBDD) *Soit f une fonction booléenne construite sur les variables $x_1, \ldots, x_n$ et soit $<$ un ordre total sur ces variables. Alors il existe un unique ROBDD codant f et respectant $<$.*

Ce théorème permet de tirer immédiatement plusieurs conséquences. On peut ainsi tester en temps constant si une formule codée par un ROBDD est une tautologie (il serait alors réduit à la feuille 1) ou si elle est satisfiable (il serait différent de la feuille 0).

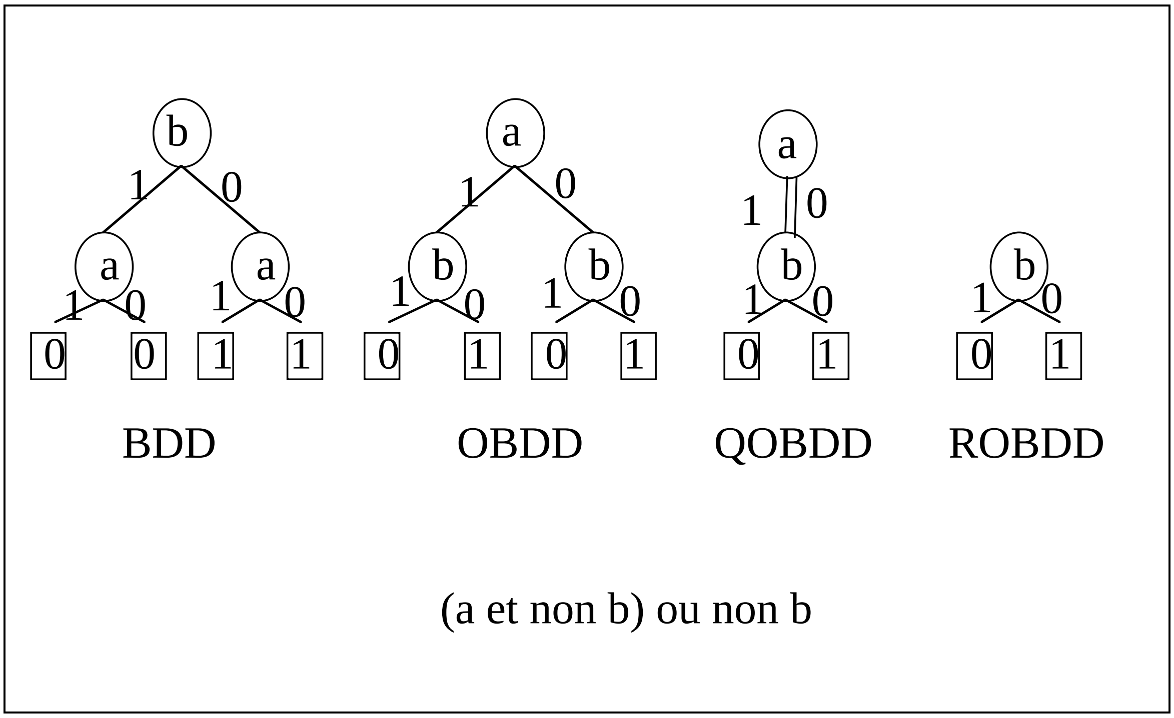

FIG. 6.1 - *Schématisation des BDD, OBDD, QOBDD et ROBDD*

6.2.2 Ordonnancement des variables dans les ROBDD

On a vu que la définition d'un ROBDD dépendait de l'ordre total donné sur les variables. Madre et Coudert ([13]) ont montré qu'il existait des fonctions dont le ROBDD était de taille polynômiale avec un certain ordre et de taille exponentielle avec un autre ordre. Il existe, de plus, des fonctions qui n'admettent pas de ROBDD de taille polynômiale, quel que soit l'ordre sur les variables (en fait, la plupart des fonctions n'admettent pas de ROBDD de taille polynômiale, même pour l'ordre optimal sur les variables).

Cependant, le problème de trouver un ordre optimal pour le ROBDD est NP-Dur et on ne connaît pas d'algorithme ayant une complexité inférieure à $O(3^n)$. Pour avoir un bon ordre sur les ROBDD destinés à coder des fonctions caractéristiques d'ensembles, Madre et Coudert conseillent de prendre un ordre sur les variables tel que :

$$S_{k-1} < O_k < S_k < O_{k+1}$$

Ce devrait permettre d'effectuer des opérations plus simples lors des traitement des ROBDD codant des ensembles, comme par exemple filtrer les ensembles contenant une certaine variable.

6.3 Nouveaux algorithmes de calcul de résolvant

Dans les deux sections qui suivent, nous allons donner des algorithmes permettant de calculer le résolvant de manière implicite, c'est-à-dire que le résultat du calcul sera la fonction caractéristique des clauses présentes dans le résolvant. Tout d'abord, on ne tient pas compte des subsomptions présentes dans le résolvant final. Le cas des subsomptions est traité dans la section 6.3.7. Après avoir défini un algorithme grandement inspiré de celui présenté par Madre et Coudert, nous en définissons un dans cette partie travaillant directement sur le ROBDD de la fonction caractéristique de l'ensemble des clauses de départ. Dans la section suivante, nous donnons un algorithme sous forme d'un ROBDD, le calcul de résolvant étant alors réduit à des opération logiques (opérateurs $\vee$, $\wedge$ et *ite*) sur plusieurs ROBDD.

6.3.1 Calcul complet d'un résolvant

Le calcul de résolvant sur une variable peut se découper en plusieurs étapes. La complexité de ce calcul vient de la distribution des clauses contenant positivement et négativement la variable sur laquelle on effectue la coupure. On peut définir plus formellement le calcul de résolvant comme suit. Soit la fonction $Clauses(f)$ donnant l'ensemble des clauses codées par la fonction caractéristique f, le calcul de résolvant sur la variable v_k peut être écrit comme :

$$Resolvant_{v_k}(f) \; = \; Distribue(\; Elimine_{v_k}(f \wedge f_{v_k}),\; Elimine_{\overline{v_k}}(f \wedge f_{\overline{v_k}})\;) \;\cup\; Clauses(f \;\wedge\; \neg f_{v_k} \;\wedge\; \neg f_{\overline{v_k}}\;)$$

où f_{v_k} est la fonction caractéristique de l'ensemble des clauses contenant la variable v_k, et $Elimine_{v_k}(f)$ renvoie l'ensemble des clauses codées par f auquelles on a enlevé la variable v_k.

Le résolvant sur n variables est simplement défini avec n appels récursifs de $Resolvant_{v_{i \leq n}}$. On voit tout de suite dans cette définition que seule la distribution des clauses pose problème, dès lors que ces dernières ne sont définies que par la fonction caractéristique de l'ensemble auquel elles appartiennnent.

D'autre part, en lisant cette définition, on peut dire que le principal problème du calcul de résolvant vient de la distribution des deux ensembles de clauses. C'est surtout ce point que l'on développe dans ce chapitre. Dans les sections suivantes, on met donc l'accent sur un nouvel algorithme de distribution des clauses basé sur leur représentation par fonction caractéristique.

6.3.2 Élimination des variables avant coupure

Il est clair que l'élimination de l'occurrence d'une variable dans un ensemble de clauses est une opération simple, même si cet ensemble est codé par un ROBDD. On peut écrire que

$$Elimine_{v_k}(f) = (f \wedge f_{O_k S_k}) \vee (f \wedge f_{O_k \overline{S_k}}) \vee (f \wedge f_{\overline{O_k}})$$

Cette élimination peut donc très bien se faire par opérations logiques entre ROBDD, ou directement comme opération sur le graphe représenté par le ROBDD.

6.3.3 Modification des algorithmes existants

Il est possible, dans une première approche, de définir de nouveaux algorithmes de calcul de résolvant en modifiant simplement les algorithmes de calcul d'ensembles d'impliquants premiers. En prenant comme base l'algorithme initial de Madre et Coudert proposé en 1994, on peut assez vite trouver des variantes de ces derniers pouvant s'appliquer au cas présent.

Algorithme de Madre et Coudert pour le calcul d'impliquants premiers

Dans [13], les auteurs décrivent récursivement l'ensemble $Prime(f)$ des impliquants premiers de la fonction booléenne f définie de $\{0,1\}^n$ dans $\{0,1\}$ par :

$$\begin{aligned}
Prime(0) =& \emptyset \\
Prime(1) =& \{\epsilon\} \\
Prime(f) =& Prime(\; f_{x_k} \wedge f_{\overline{x_k}}\;) \;\cup \\
& \{\overline{x_k}\} \times (Prime(\; f_{\overline{x_k}}\;) \setminus Prime(\; f_{x_k} \wedge f_{\overline{x_k}}\;)\;) \;\cup \\
& \{x_k\} \times (Prime(\; f_{x_k}\;) \setminus Prime(\; f_{x_k} \wedge f_{\overline{x_k}}\;)\;)
\end{aligned}$$

Si cette fonction booléenne f est mise sous FNS et codée par un ROBDD, on peut appliquer les équations récursives suivantes sur le ROBDD de f pour obtenir directement le MétaProduit de ses impliquants premiers. C'est là un des éléments clés de la puissance de l'approche de Madre et Coudert.

$$\begin{aligned}
Prime_k(0) &= 0 \\
Prime_k(1) &= \neg \left(\bigvee_{j=1}^{n} O_j \right) \\
Prime_k(f) &= \Delta(O_k, \Delta(S_k, Prime_{k+1}(f_{x_k}) \wedge \neg Prime_{k+1}(f_{\overline{x_k}} \wedge f_{x_k}), \\
&\qquad Prime_{k+1}(f_{\overline{x_k}}) \wedge \neg Prime_{k+1}(f_{\overline{x_k}} \wedge f_{x_k})), \\
&\quad Prime_{k+1}(f_{\overline{x_k}} \wedge f_{x_k}))
\end{aligned}$$

Modification de l'algorithme initial

En utilisant le théorème 2, il vient rapidement une description récursive de l'ensemble $Resolvant(f)$ des clauses présentes dans le résolvant de la fonction booléenne f. Il suffit pour cela de modifier $Prime$ pour ne pas construire les clauses contenant les variables à éliminer.

$$\begin{aligned}
Resolvant(0) &= \emptyset \\
Resolvant(1) &= \{\epsilon\} \\
Resolvant(f) &= \begin{cases} Resolvant(\ f_{x_k} \wedge f_{\overline{x_k}}\ ) & \text{Si } x_k \text{ doit être éliminé} \\ \\ \begin{pmatrix} Resolvant(\ f_{x_k} \wedge f_{\overline{x_k}}\ ) \ \cup \\ \{\overline{x_k}\} \times Resolvant(\ f_{\overline{x_k}}\ ) \ \cup \\ \{x_k\} \times Resolvant(\ f_{x_k}\ ) \end{pmatrix} & \text{Sinon} \end{cases}
\end{aligned}$$

Dans cette description récursive, on note que la partie garantissant la primalité des ensembles d'impliquants est absente (on a bien précisé au départ qu'on ne tenait pas compte des subsomptions dans cette première approche). Si ceci peut être vu comme un gain de rapidité, il ne faut pas oublier qu'il faudra tout de même filtrer les clauses subsumées dans la suite du calcul. Le gain n'est donc pas si clair. Cependant, par rapport à la définition de $Prime$, l'utilisation de $Resolvant$ réduit trivialement le nombre d'appels récursifs d'un facteur dépendant du nombre de variables à éliminer.

Limitations de cette approche

Même si cette approche promet de bons résultats dans l'analyse d'arbres de défaillance ou de systèmes physiques (domaines de prédilection de l'algorithme de Madre et Coudert), on est cependant assez proche de l'algorithme initial. En dehors du fait qu'on risque de ne pas avoir un algorithme suffisamment original pour espérer avoir des résultats sensiblement différents de l'algorithme initial, on risque surtout de retrouver une des caractéristiques les plus fâcheuses de cet algorithme : il faut pouvoir réécrire la formule, initialement sous FNC, sous FNS. Or, ceci ne peut clairement pas être envisageable pour les formules dont la satisfiabilité (ou l'insatisfiabilité) est difficile à prouver ou pour toutes celles qui représentent encore aujourd'hui des challenges au problème SAT, parce le passage de la formule de FNC à FNS répond aux questions de satisfiabilité et dínsatisfiabilité. . On a bien vu en 4.3 que l'algorithme de calcul de résolvant pouvait être appliqué avec succès à ce type de problèmes comme précompilation des formules à traiter. Les bons résultats des algorithmes de type Madre et Coudert viennent intuitivement du fait que les exemples difficiles à traiter (dans leur domaine de formules modélisant des systèmes physiques) contiennent énormément d'impliquants premiers. Or, on peut très bien imaginer des formules contenant un nombre raisonnable d'impliquants premiers, mais dont ceux-ci seraient *difficiles* à obtenir, en terme de résolution par exemple, et non plus en terme de quantité de ces derniers.

Les prochaines approches que nous allons définir n'utilisent plus en entrée la formule passée sous forme de ROBDD, mais le ROBDD codant la fonction caractéristique des clauses composant cette même formule sous forme normale conjonctive. En procédant ainsi, on peut espérer traiter, du moins en partie, des formules dont on ne sait pas prouver en temps raisonnable ni la satisfiabilité ni l'insatisfiabilité.

6.3.4 Calcul de résolvant par traitement d'un ROBDD

Dans cette partie, nous présentons un algorithme travaillant sur le ROBDD codant l'ensemble des clauses de la formule de départ. En se basant sur la section 6.3.1, le point principal de l'algorithme à trouver est la distribution entre les deux ensembles de clauses où apparaît la variable à éliminer positivement et négativement. Ensuite, il n'y a plus que des manipulations ensemblistes, facilement et efficacement codables par des opérations sur les ROBDDD.

Distribution de deux ensembles de clauses

THÉORÈME 5 *L'ensemble $Distribue(g,h)$ des clauses composant la distribution des deux ensembles de clauses EC_g et EC_h dont les fonctions caractéristiques sont respectivement g et h est récursivement décrit par :*

$$
\begin{aligned}
Distribue(0,0) =& \emptyset \\
Distribue(1,1) =& \{\epsilon\} \\
Distribue(1,0) =& \{\epsilon\} \\
Distribue(0,1) =& \{\epsilon\} \\
Distribue(g,h) =& Distribue(\ (g \wedge \overline{O_k}),\ (h \wedge \overline{O_k})) \ \cup \\
& \{\overline{x_k}\} \times Distribue(\ g \wedge O_k \wedge \overline{S_k},\ h \wedge O_k \wedge \overline{S_k}) \ \cup \\
& \{\overline{x_k}\} \times Distribue(\ (g \wedge \overline{O_k}),\ h \wedge O_k \wedge \overline{S_k}) \ \cup \\
& \{\overline{x_k}\} \times Distribue(\ g \wedge O_k \wedge \overline{S_k},\ (h \wedge \overline{O_k})) \ \cup \\
& \{x_k\} \times Distribue(\ g \wedge O_k \wedge S_k,\ h \wedge O_k \wedge S_k) \ \cup \\
& \{x_k\} \times Distribue(\ (g \wedge \overline{O_k}),\ h \wedge O_k \wedge S_k) \ \cup \\
& \{x_k\} \times Distribue(\ g \wedge O_k \wedge S_k,\ (h \wedge \overline{O_k}))
\end{aligned}
$$

Preuve : La preuve est donnée en Annexe A.2[1].

Il ne reste plus qu'à substituer des opérations logiques sur les MétaProduits à la place des opérations ensemblistes, et on obtient un algorithme récursif qui construit le ROBDD du MétaProduit de la distribution des clauses à partir des ROBDD des MétaProduits des deux ensembles de clauses de départ. On obtient les équations récursives suivantes pour la distribution des clauses :

$$
\begin{aligned}
& Distri_k(0,0) = 0 \\
& Distri_k(1,0) = Distri_k(0,1) = Distri_k(1,1) = \neg \left(\bigvee_{j=1}^{n} O_j \right) \\
& Distri_k(g,h) = \Delta(O_k, \\
& \qquad \Delta(S_k, Distri_{k+1}(g_{O_k S_k}, h_{O_k S_k}) \ \vee \ Distri_{k+1}(g_{\overline{O_k}}, h_{O_k S_k}) \ \vee \ Distri_{k+1}(g_{O_k S_k}, h_{\overline{O_k}}), \\
& \qquad\quad Distri_{k+1}(g_{O_k \overline{S_k}}, h_{O_k \overline{S_k}}) \ \vee \ Distri_{k+1}(g_{\overline{O_k}}, h_{O_k \overline{S_k}}) \ \vee \ Distri_{k+1}(g_{O_k \overline{S_k}}, h_{\overline{O_k}})) \\
& \qquad Distri_{k+1}(g_{\overline{O_k}}, h_{\overline{O_k}}))
\end{aligned}
$$

Si on évalue $Distri_1$ appliqué à deux ROBDD codant des ensembles de clauses, on obtient directement le ROBDD codant l'ensemble des clauses représentant la distribution des deux ensembles de clauses de départ.

6.3.5 Algorithmes par opérations logiques entre ROBDD

Les algorithmes que nous venons de présenter travaillent directement sur les ROBDD codant les ensembles de clauses. Or, la puissance des ROBDD permet en fait d'exprimer les algorithmes eux-même sous forme de ROBDD. En procédant ainsi, on peut espérer obtenir une distribution de clauses avec de simples opérations classiques sur les ROBDD. Encore une fois, il ne nous est pas possible de comparer l'approche précédente avec celle que nous allons présenter maintenant. Néanmoins, l'avantage de cette dernière approche est qu'elle travaille par compilation d'un ROBDD codant la distribution. Une fois ce ROBDD dépendant du nombre de variables des ensembles de clauses à distribuer compilé, la distribution elle-même se réduit, comme on va le voir, à la substitution de deux variables du ROBDD par les fonctions caractéristiques des ensembles à distribuer, cette substitution étant une opération classique sur les ROBDD (application de l'opérateur *ite*).

6.3.6 Distribution implicite de deux ensembles de clauses

L'idée de base pour trouver l'algorithme est une définition en intention de la présence d'une clause dans l'ensemble des clauses codant le résolvant, définie d'après les définitions en intention des deux ensembles de clauses à distribuer.

1. (page (VI))

Soit E l'ensemble des clauses issues de la distribution des deux ensembles de clauses A et B. Si c, c_A et c_B sont des clauses et $\mu(c)$, $\mu(c_A)$, $\mu(c_B)$ leur MétaSomme respective, l'équivalence suivante nous guide dans la conception d'un tel algorithme :

$$c \in E \Leftrightarrow \left\{ \begin{array}{l} \exists c_A \in A, \exists c_B \in B \text{ tels que} \\ \qquad \forall o_i s_i \in \mu(c), \\ \qquad\qquad \begin{cases} \begin{pmatrix} (o_i s_i \in \mu(c_A)) \ \wedge \ (o_i \overline{s_i} \notin \mu(c_B)) \\ \vee\ (o_i \overline{s_i} \notin \mu(c_A)) \ \wedge \ (o_i s_i \in \mu(c_B)) \end{pmatrix} & \text{si } o_i s_i \\ \begin{pmatrix} (o_i \overline{s_i} \in \mu(c_A)) \ \wedge \ (o_i s_i \notin \mu(c_B)) \\ \vee\ (o_i s_i \notin \mu(c_A)) \ \wedge \ (o_i \overline{s_i} \in \mu(c_B)) \end{pmatrix} & \text{si } o_i \overline{s_i} \\ (o_i \notin \mu(c_A)) \ \wedge \ (o_i \notin \mu(c_B)) & \text{sinon } (\overline{o_i}) \end{cases} \end{array} \right\}$$

Ce qui signifie, moins formellement, qu'une clause c est dans la distribution de deux ensembles de clauses si et seulement si on peut trouver une clause c_A dans A et une clause c_B dans B telle que tous les littéraux de c sont présents dans au moins une des deux clauses (et ni c_A, ni c_B ne contient son opposé) et que toutes les variables qui ne sont pas dans c ne sont pas non plus ni dans c_A, ni dans c_B.

On voit dans une telle définition que le problème est de trouver c_A et c_B de manière efficace. Il suffit pour cela de construire au fur et à mesure les sous-ensembles de A et B susceptibles de contenir c_A et c_B. Voyons comment cela peut être fait dans une seconde approche.

Soit la fonction $Distri_i(A, B)$ pour A et B fixés, définie sur $([o_i \dots o_n] \times [s_i \dots s_n])$ et donnant vrai si et seulement si $[o_i s_i \dots o_n s_n]$ est le MétaProduit de la fin au rang i d'une clause distribuée sur $A \times B$ (on appelle *fin d'une clause* c au rang i la restriction des $o_k s_k$ composant son Métaproduit au sous ensemble $o_{k_{k \geq i}} s_{k_{k \geq i}}$). Le principe de récurrence, donné pour la présence du littéral x_i est le suivant : une clause est la fin au rang i d'une clause distribuée sur A et B si :

- x_i fait partie de la clause ET la fin au rang $i+1$ de cette clause est la distribution de deux clauses sur les ensembles $A_{Contenant\ x_i}$ et $B_{Ne\ Contenant\ pas\ \overline{x_i}}$, OU
- x_i fait partie de la clause ET la fin au rang $i+1$ de cette clause est la distribution de deux clauses sur les ensembles $A_{Ne\ Contenant\ pas\ \overline{x_i}}$ et $B_{Contenant\ x_i}$, OU
- x_i n'est pas dans la clause ET la fin au rang $i+1$ de cette clause est la distribution de deux clauses sur les ensembles $A_{Ne\ Contenant\ pas\ x_i\ ni\ \overline{x_i}}$ et $B_{Ne\ Contenant\ ni\ x_i\ ni\ \overline{x_i}}$

Une telle fonction est décrite formellement figure 6.2.

THÉORÈME 6 (Distribution par opération logique) $[Distri_1(A, B)](c)$ *renvoie vrai si et seulement si le MétaProduit de la clause c fait partie de la distribution des deux ensembles de clauses respectifs A et B.*

Preuve : La preuve est donnée en Annexe A.3 [2].

Encore une fois, les équations récursives correspondantes, figure 6.3 s'écrivent assez facilement d'après la définition de la figure 6.2, en définissant le ROBDD compilé au fur et à mesure. Quelques difficultés méritent cependant qu'on précise la transformation. Ainsi, on ne peut pas se contenter de $2n$ variables (les $O_i S_i$) pour coder le ROBDD compilé. Il faut pouvoir trouver un ROBDD dépendant de A et B. Pour cela, on utilise une opération classique sur les ROBDD, qui est la *substitution d'une variable par une formule.*

Si f est le ROBDD codant une formule contenant la variable x, alors le ROBDD codant la formule f dans laquelle x est substituée par une fonction g est le ROBDD $f \wedge ite(x, g, \neg g)$.

2. (Page (IX))

$$Distri_i(A,B)\ :\ ([o_i \dots o_n] \times [s_i \dots s_n])\ \mapsto\ Bool$$

$$Distri_i(A,B) = \begin{cases} \begin{pmatrix} Distri_{i+1}(\{x \in A \mid \{o_i s_i\} \in x\}, \{x \in B \mid \{o_i \overline{s_i}\} \notin x\}) \\ \vee\, Distri_{i+1}(\{x \in A \mid \{o_i \overline{s_i}\} \notin x\}, \{x \in B \mid \{o_i s_i\} \in x\}) \end{pmatrix} & \text{si } o_i s_i \\ \begin{pmatrix} Distri_{i+1}(\{x \in A \mid \{o_i \overline{s_i}\} \in x\}, \{x \in B \mid \{o_i s_i\} \notin x\}) \\ \vee\, Distri_{i+1}(\{x \in A \mid \{o_i s_i\} \notin x\}, \{x \in B \mid \{o_i \overline{s_i}\} \in x\}) \end{pmatrix} & \text{si } o_i \overline{s_i} \\ Distri_{i+1}(\{x \in A \mid \{\overline{o_i}\} \in x\}, \{x \in B \mid \{\overline{o_i}\} \in x\}) & \text{sinon } (\overline{o_i}) \end{cases}$$

$$Distri_n(A,B) = \begin{cases} \begin{pmatrix} ((\{x \in A \mid \{o_n s_n\} \in x\} \neq \emptyset) \\ \quad \wedge (\{x \in B \mid \{o_n \overline{s_n}\} \notin x\} \neq \emptyset)) \\ \vee((\{x \in A \mid \{o_n \overline{s_n}\} \notin x\} \neq \emptyset) \\ \quad \wedge (\{x \in B \mid \{o_n s_n\} \in x\} \neq \emptyset)) \end{pmatrix} & \text{si } o_n s_n \\ \begin{pmatrix} ((\{x \in A \mid \{o_n \overline{s_n}\} \in x\} \neq \emptyset) \\ \quad \wedge (\{x \in B \mid \{o_n s_n\} \notin x\} \neq \emptyset)) \\ \vee((\{x \in A \mid \{o_n s_n\} \notin x\} \neq \emptyset) \\ \quad \wedge (\{x \in B \mid \{o_n \overline{s_n}\} \in x\} \neq \emptyset)) \end{pmatrix} & \text{si } o_n \overline{s_n} \\ (\{x \in A \mid \{\overline{o_n}\} \in x\} \neq \emptyset) \\ \vee (\{x \in B \mid \{\overline{o_n}\} \in x\} \neq \emptyset) & \text{sinon } (\overline{o_n}) \end{cases}$$

FIG. 6.2 - *Définition de $Distri_1(A,B)$*

Pour autoriser cette substitution dans le ROBDD compilé, on ajoute les deux variables a_1 et b_1. Ces deux variables seront substituées par les fonctions caractéristiques des deux ensembles respectifs à distribuer. Cette technique de substitution est utilisée aussi pendant la compilation du ROBDD. Chaque $BDDistri_i$ contient donc deux variables supplémentaires a_i et b_i pour permettre une telle manipulation.

$$\begin{aligned} BDDistri_i = \Delta(O_i,& \\ &\Delta(S_i, (BDDistri_{i+1} \wedge ite(a_{i+1}, a_i \wedge O'_i \wedge S'_i, 0) \wedge ite(b_{i+1}, b_i \wedge \neg(O'_i \wedge \overline{S'_i}), 0)) \vee \\ &\qquad (BDDistri_{i+1} \wedge ite(a_{i+1}, a_i \wedge O'_i \wedge \overline{S'_i} \wedge ite(b_{i+1}, b_i \wedge \neg(O'_i \wedge S'_i), 0))), \\ &\quad (BDDistri_{i+1} \wedge ite(a_{i+1}, a_i \wedge O'_i \wedge \overline{S'_i}), 0) \wedge ite(b_{i+1}, b_i \wedge \neg(O'_i \wedge S'_i), 0)) \vee \\ &\qquad (BDDistri_{i+1} \wedge ite(a_{i+1}, a_i \wedge O'_i \wedge S'_i \wedge ite(b_{i+1}, b_i \wedge \neg(O'_i \wedge \overline{S'_i}), 0))), \\ &(BDDistri_{i+1} \wedge ite(a_{i+1}, a_i \wedge \overline{O'_i}, 0) \wedge ite(a_{i+1}, a_i \wedge \overline{O'_i}, 0)) \\ BDDistri_n = \Delta(O_n,& \\ &\Delta(S_n, (a_n \wedge O'_n \wedge S'_n) \vee (b_n \wedge \neg(O'_n \wedge \overline{S'_n})), \\ &\quad (a_n \wedge O'_n \wedge \overline{S'_n}) \vee (b_n \wedge \neg(O'_n \wedge S'_n))), \\ &(a_n \wedge O_n) \vee (b_n \wedge O_n)) \end{aligned}$$

FIG. 6.3 - *Définition du ROBDD à compiler permettant la distribution des ensembles de clauses*

Enfin, il faut prendre garde de renommer les ROBDD codant les ensembles A et B pour

6.3.7 Traitement des subsomptions

Si on calcule les clauses du résolvant de manière implicite, le nombre de clauses manipulées n'est plus une limite intrinsèque de l'algorithme. Cependant, il ne faut pas pour autant fournir un résolvant dont les clauses subsumées n'auraient pas été traitées. On peut en effet vérifier avec l'approche classique du calcul

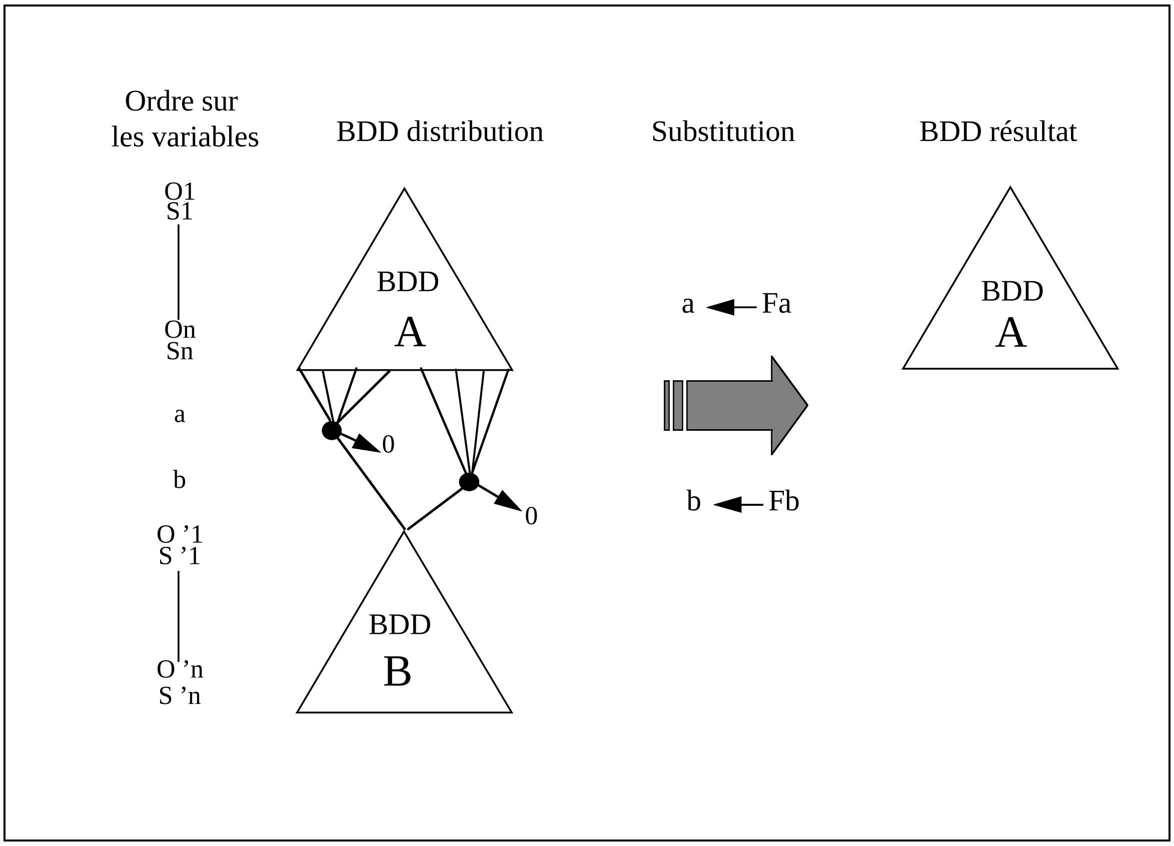

Fig. 6.4 - *Schéma de distribution par opération logique*

de résolvant que le nombre d'éliminations de clauses par subsomptions est important dans les exemples codant des systèmes physiques (voir chapitre tests de la méthode classique). De plus, on ne peut guère espérer voir le nombre de clauses subsumées diminuer avec les coupures, car il a tendance, comme on peut le voir, à croître exponentiellement.

Soit une coupure sur la variable v. Soit C_v et $C_{\overline{v}}$ l'ensemble des clauses contenant v sous forme positive et négative respectivement. Si il existe une clause c_1 de C_v subsumée par une autre clause c_2 de C_v avant la coupure, il est facile de vérifier que le résolvant sur v contient de l'ordre de $Card(C_{overlinev})$ clauses subsumées (ces clauses sont produites par la distribution de c_1 sur $C_{\overline{v}}$ et chacune d'elles est subsumée par une des clauses produites par la distribution de c_2 sur $C_{\overline{v}}$).

Ne pas tenir compte des subsomptions peut vite rendre le résultat du calcul illisible. Il faut donc pouvoir traiter le ROBDD codant le résultat du calcul pour en éliminer toute clause subsumée. Ce calcul peut malgré tout être efficacement repoussé jusqu'à la fin de l'élimination des variables, permettant de n'être exécuté qu'une seule et unique fois (la difficulté de l'approche classique vient, comme on l'a vu dans un chapitre précédent, du traitement des subsomptions dans la formule produite après chaque coupure, seule possibilité pour tenter de limiter l'explosion combinatoire du nombre de clauses).

Encore une fois, ceci n'est possible que parce que le nombre de clauses produites par coupures n'est plus une limite au calcul. On peut donc retarder sans risque de perte de coût ce traitement jusqu'à la fin de la dernière coupure.

Traitement sur le ROBDD codant le résolvant

En prenant exemple sur la définition de l'ensemble des impliquants premiers d'une fonction booléenne mise sous forme de ROBDD proposée dans [13], on peut décrire la fonction $NoSub$ qui filtre les clauses non subsumées d'un ensemble de clauses de départ.

Soit f la fonction caractéristique d'un ensemble de clause. On peut décrire récursivement l'ensemble $NoSub(f)$ des clauses non subsumées de l'ensemble des clauses caractérisé par la fonction booléenne f par :

$$\begin{aligned} NoSub(0) =& \emptyset \\ NoSub(1) =& \{\epsilon\} \\ Nosub(f) =& NoSub(f \wedge \overline{O_k}) \ \cup \\ & \{\overline{x_k}\} \times (NoSub(f \wedge O_k \wedge \overline{S_k}) \setminus NoSub(f \wedge \overline{O_k})) \ \cup \\ & \{x_k\} \times (NoSub(f \wedge O_k \wedge S_k) \setminus NoSub(f \wedge \overline{O_k})) \cup \end{aligned}$$

Les équations récursives basées sur ces définitions sont proches de celles données dans l'algorithme de Madre et Coudert, et on peut définir directement le ROBDD codant l'ensemble des clauses non subsumées d'après le ROBDD codant les clauses de f.

$$\begin{aligned} NoSub_k(0) &= 0 \\ NoSub_k(1) &= \neg \left(\bigvee_{j=1}^{n} O_j \right) \\ NoSub_k(f) &= \Delta(O_k, \Delta(S_k, NoSub_{k+1}(f \wedge O_k \wedge S_k) \wedge \neg NoSub_{k+1}(f \wedge \overline{O_k}), \\ &\qquad NoSub_{k+1}(f \wedge O_k \wedge \overline{S_k}) \wedge \neg NoSub_{k+1}(f \wedge \overline{O_k})), \\ &\quad NoSub_{k+1}(f \wedge \overline{O_k})) \end{aligned}$$

Comme précédemment, on peut aussi définir un filtre sur l'ensemble des clauses grâce à un ROBDD. Il suffit de noter que l'on peut exprimer la fonction caractéristique $f_{EC'}$ de l'ensemble des clauses non subsumées entre elles de cette même formule :

$$c \in f_{EC'} \Leftrightarrow (c \in f_{EC}) \wedge (\forall c' \neq c \in f_{EC}, (\forall (o_i s_i) \in \mu(c), \overline{o_i} \rightarrow (\overline{o_i} \in \mu(c')))))$$

Ce qui n'est qu'une formalisation du fait qu'une clause c est dans $f_{EC'}$ si et seulement si elle est dans f_{EC} et qu'aucune autre clause de f_{EC} contient uniquement un sous-ensemble strict des littéraux composant c.

Encore une fois, on peut transformer cette définition en un ROBDD dépendant du ROBDD codant f_{EC}, en réduisant au fur et à mesure du calcul l'ensemble des clauses pouvant subsumer celles passées en argument. Même si nous avons mené cette partie du travail à terme, la formalisation de la compilation d'un ROBDD permettant le filtrage des clauses est lourde à mettre en place et n'ajouterait rien aux voies ouvertes dans ce rapport. De plus, on peut penser que dans le cas du filtrage des clauses subsumées, l'algorithme que nous venons de présenter travaillant sur le ROBDD des clauses à filtrer peut avoir une complexité satisfaisante, même s'il faudrait, encore une fois, mener de plus vastes expérimentations dans cet axe.

6.3.8 Compilation d'un calcul de résolvant *universel*

Nous venons de voir comment il était possible de faire une coupure de manière implicite. Il serait intéressant de définir un calcul de résolvant entièrement grâce à un ROBDD, codant par exemple l'élimination de k variables sur n composants la formule de départ.
Dans le cas d'une telle réalisation, si celle compilation était possible, on obtiendrait un calcul de résolvant universel, applicable à l'identique pour toutes les formules comprenant n variables et dont le résolvant est sur k variables. Malheureusement, nous n'avons encore aucun indice nous permettant de savoir si une telle méthode pourrait être appliquée.

Chapitre 7

Perspectives

En essayant d'appliquer l'algorithme à plusieurs domaines de l'informatique, en prenant le temps de mener une expérimentation de l'algorithme, en tentant de contourner les difficultés contre lesquelles le programme butait, en permettant l'expression de nouveaux algorithmes de calcul de résolvant, nous avons été amené à faire des choix rapides et définitifs pour permettre une étude soignée de certaines voies que nous jugions primordiales avant toute poursuite. En faisant ces choix, naturellement, nous avons volontairement mis de coté nombre d'approches qui nous auraient pourtant semblé intéressantes. L'approfondissement de celles-ci pourrait être fait dans une suite logique du travail que nous venons de présenter. Nous donnons brièvement dans ce chapitre, en cette vue, plusieurs axes qu'il nous semble intéressant d'étudier dans la suite.

7.1 Extensions du programme

On peut tout d'abord présenter une extension simple au programme, mais qui peut permettre d'appréhender une nouvelle classe de problèmes sans passer par une compilation des formules. C'est le passage de l'algorithme en Attribut-Comparateur-Valeur. Pour cela, il suffit de noter que la coupure entre deux clauses est en fait l'application directe de la transitivité de l'opérateur d'implication. En effet, $(x \vee A)$ est équivalent à $(\neg A \rightarrow x)$ et $(\neg x \vee B)$ se réécrit en $(x \rightarrow B)$. Par application de la transitivité de l'opérateur $\rightarrow$, on déduit de $(\neg A \rightarrow x)$ et $(x \rightarrow B)$ que $(\neg A \rightarrow B)$, qui est bien équivalent à $(A \vee B)$.
En appliquant cette transitivité aux attributs-comparateurs-valeurs directement, on peut facilement définir une coupure entre deux clauses d'ordre O^+, avec la même complexité qu'en propositionnel. Ceci pourrait permettre une accéleration du programme dans le cas de compilation de bases de connaissances, qui s'expriment plus facilement dans un ordre comme 0^+ plutôt qu'en propositionnel.

7.1.1 Des opérations directes sur les Tries

On peut espérer trouver des algorithmes permettant de trouver la distribution de deux ensembles de clauses, chaque ensemble étant représenté par un Trie, sans passer par une énumération des deux ensembles de clauses. Ceci peut représenter une clé pour obtenir des opérations de coupures beaucoup plus rapides que celles que nous avons implémenté. Même si, rappelons-le, nous avons travaillé dans cette voie, nous n'avons pas eu le temps de finir cette approche.

7.2 L'approche hybride

La première optimisation que l'on pourrait proposer est certainement l'application des principes de manipulation des Tries pour DPL. Dans [26], les auteurs proposent un algorithme DPL basé directement sur les Tries, qui facilite la propagation unitaire et qui a permis aux auteurs de résoudre des problèmes encore ouverts en 1994. On pourrait raisonnablement penser adapter ces méthodes lors du passage de l'algorithme hybride en période DPL et augmenter ainsi la rapidité des propagations unitaires, permettant des simplifications rapides de la formule traitée.

7.2.1 De meilleures heuristiques

On en a parlé au moment de l'écriture des heuristiques : pour déterminer le moment de passage de l'algorithme hybride en DPL, pour déterminer une durée de passage optimale et pour déterminer les variables permettant une simplification maximale de la formule, il reste encore du travail à faire et des heuristiques à tester. Celle que nous avons implémentée n'a donc pas pu être comparée avec d'autres, mais nous avons défini le principe de telles heuristiques : minimiser la taille des formules présentes aux feuilles de l'arbre de Shannon.

7.2.2 Déséquilibrer l'arbre de Shannon

On peut aller encore plus loin dans l'approche *hybride-finale*, en déséquilibrant volontairement l'arbre de décomposition de Shannon, pour obtenir sur chaque feuille des calculs de résolvants les plus simples possibles. Ainsi, il existe des formules pour lesquelles une affectation d'une variable à 1 élimine la moitié des clauses, mais dont l'affectation à 0 ne touche pratiquement pas de clauses. On peut imaginer alors des branches de profondeur 1 et des branches de profondeur 10, qui déboucheraient toutes sur approximativement le même nombre de clauses. On serait alors restreint à la seule approche *hybride-final*, en laissant de côté *hybride-borné*, mais on pourrait choisir un nombre maximal de feuilles dans l'arbre, et non plus une profondeur maximale de l'arbre, ce qui peut permettre une plus grande flexibilité dans le choix des variables et une optimisation encore plus fine du nombre de feuilles à traiter. Il faudrait alors trouver de nouvelles heuristiques adéquates à cette nouvelle formulation de *hybride-borné*. Le principe de cette extension est schématisé figure 7.1.

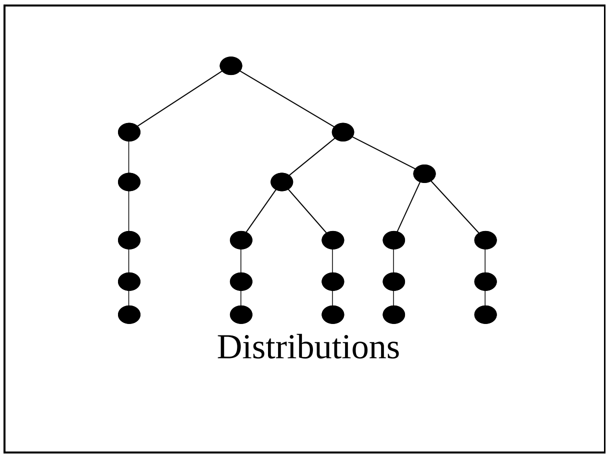

FIG. 7.1 - *Désequilibrer l'arbre de Shannon*

7.2.3 Parallélisation du calcul de résolvant

En passant en approche hybride, il vient naturellement l'idée de paralléliser le calcul. On doit en effet calculer des résolvants, dans le cas de *hybride-final* sur chaque feuille de l'arbre de Shannon. Si on exclut la distribution finale des clauses, on obtient un algorithme dont le coût est divisé exactement pas le nombre de processeurs mis à disposition. Cependant, il serait intéressant de poursuivre l'investigation pour tenter de trouver une distribution de clauses parallélisable, ce qui permettrait alors d'exprimer totalement le calcul de résolvant de manière parallèle. L'algorithme DPL est par essence parallélisable, mais proposer DP en parallèle est un problème encore ouvert.

7.2.4 Optimisations

On peut aussi gagner énormément en optimisant encore les structures de données pour prendre en considération les simplifications que l'on peut être amené à faire dans l'approche hybride. Ainsi, on peut tenter de trouver un opérateur sur le Trie des clauses permettant de renvoyer le Trie des clauses simplifiées par une certaine affectation de variables. D'ailleurs, dans [26], l'auteur utilise un Trie pour un algorithme DP, pour faciliter les propagations unitaires (on obtient alors un coût sous-linéaire), nombreuses dans DP.

7.3 Tests et approfondissements de la méthode implicite

Nous avons proposé des algorithmes dans le dernier chapitre sans les implémenter. Le premier travail à faire dans la suite de ce rapport consiste bien entendu à tester pratiquement ces algorithmes, pour comprendre leur comportement face aux algorithmes traditionnels de calcul de résolvant que l'on a pu étudier jusque là. Il reste cependant des zones d'ombre communes à toutes les méthodes de calcul implicites. Jusqu'à aujourd'hui, en effet, rien n'a été fait pour tenter de comprendre pourquoi l'utilisation de ROBDD permet de traiter des exemples concrets avec un gain énorme par rapport aux méthode explicites, plus traditionnelles. Il serait intéressant de voir si on ne peut pas exprimer des caractéristiques sur les formules concrètes permettant de comprendre en quoi ces caractéristiques autorisent une représentation efficace par des ROBDD.

7.4 Vers une solution pour Parity32 ?

Dans [2], les auteurs posent une dizaine de *challenges* en IA. Parmi ceux-ci, le second challenge concerne un des fichiers des exemples de DIMACS, nommé Parity32. Les auteurs prétendent que le test de satisfiabilité de cette formule représente un challenge à résoudre dans un temps compris entre 2 et 5 ans. Sans vouloir prétendre résoudre ce problème par notre méthode, nous avons tout de même lancé le calcul de résolvant sur cette formule contenant plus de 10000 clauses et 3176 variables. Or, le calcul de résolvant permet une élimination assez facile de plus de 2500 variables. On arrive à obtenir une formule de quelques 5200 clauses avec 770 variables dont la preuve de satisfiabilité prouverait la satisfiabilité de Parity32. Si on continue de calculer le résolvant, le nombre de clauses monte vite et on obtient plus de 40000 clauses pour 380 variables restantes. Par curiosité, nous avons lancé des programmes de tests de satisfiabilité sur ces différents résolvants calculés. Sans succès.
Nous avons voulu aller plus loin. En nous servant de l'approche *hybride-finale*, dont la partie DPL est lancée en cas de coupure produisant plus de 10000 clauses, et dont la profondeur est fixée à 10, nous avons réussi à aller jusqu'aux feuilles de notre arbre de développement. Le programme a lancé 3 périodes DPL entrecoupées de poursuite de calcul en DP, donc nous avions 2^{10*3} feuilles dans notre arbre. Le résolvant (clause vide ou résolvant vide) de chacune de ces feuilles semble être calculable (nous en avons calculé plus de 200 sur la totalité), mais ceci reste clairement insuffisant au vue du nombre de feuilles à explorer dans le pire des cas (rappelons que dans le cas du problème SAT, il suffit de trouver un résolvant qui ne soit pas la clause vide pour démontrer la satifiabilité de la formule).
Mais, l'enseignement de ces expérimentations sur Parity32 permet de dire qu'en affectant correctement seulement ces 30 variables utilisées pour la propagation des 3 parties DPL successives, nous pouvions prouver la satisfiabilité de la formule. Ces 30 variables représentent cependant un obstacle insurmontable s'il fallait explorer les 2^{30} feuilles de l'arbre. Nous n'avons pas mené les expérimentations plus loin, mais on peut trouver sans prétention cette voie prometteuse.

Chapitre 8

Conclusion

Finalement, et on l'a vu avec certains résultats encourageants sur SAT, le calcul de résolvant, pourtant présenté en 1960, n'a pas eu jusqu'à présent de réelle chance de montrer ces qualités. Certes, celui-ci reste d'une grande complexité sur des problèmes aléatoires, mais sur ceux qui apparaissent avec une certaine structure, il peut donner comme on l'a vu de très bons résultats.

Dans ce rapport, nous avons essayé de mêler la rigueur d'une expérimentation scientifique sur un algorithme que nous avons implémenté avec le temps disponible au cours d'un stage de DEA. Il a fallu, dans un temps limité, distinguer les cas faciles des cas difficiles et comprendre ce qui rendait tel exemple pratiquable pour le programme et tel exemple irréalisable. Cerner ses points forts et ses défauts nous a permis de développer des variantes de l'algorithme initial pour essayer de contourner ou de repousser un peu les points noirs identifiés.

La contribution du travail présenté dans ce rapport est multiple. Tout d'abord, le calcul de résolvant, initialement utilisé dans le cadre de vérification d'arbres de défaillance de systèmes physiques a trouvé nombre d'applications, dont certaines, comme l'application au problème SAT, ont obtenu de meilleurs résultats que les algorithmes systématiques de référence actuels sur nombre d'exemples utilisés traditionnellement comme benchmarks. De plus, on a aussi vu que le calcul de résolvant ne se restreignait pas à une application, mais qu'il touchait à un grand nombre de problèmes classiques d'intelligence artificielle, comme la compilation de bases de connaissances. Ensuite, la première implémentation de l'algorithme a bien mis en évidence l'importance des heuristiques dans ce calcul et plusieurs d'entre elles ont été comparées intensivement face à des exemples aléatoires, ce qui peut compléter les travaux de Dechter et Rish [9, 10] qui ont surtout étudié l'une d'elles dans le cadre de la *Directional Resolution*. On peut aussi rappeler l'importance de la mise en oeuvre de structures de données compactes, comme les Tries, permettant des manipulations d'ensembles de clauses et des tests de subsomptions beaucoup plus rapides. Une troisième contribution de ce rapport a certainement été l'approche hybride pour le calcul de résolvant, permettant de limiter ou de repousser l'explosion combinatoire du nombre de clauses. En effet, il semble que l'on soit arrivé au bout des méthodes traditionnelles, chacune dévoilant ses défauts majeurs à mesure qu'on l'étudie. Pour aller plus loin et repousser encore ces exemples "bêtes noires", la seule voie qui nous est apparue est de mêler plusieurs approches différentes dans le même calcul, et permettre au programme de passer d'une approche à l'autre durant le calcul. Une telle idée semble être une des clés permettant de contourner les difficultés inhérentes à des algorithmes trop figés dans leur méthode. Enfin, le chapitre six a largement ouvert les perspectives d'un travail approfondi sur le calcul d'un résolvant implicite, en proposant deux nouveaux algorithmes de calcul de résolvants, dont le coût de calcul est indépendant du nombre de clauses manipulés, et qui peuvent aussi s'appliquer au champ d'application identifié au début du rapport. Même si aucune expérimentation n'a eu lieu par faute de temps, ce qui aurait nécessité l'implémentation des algorithmes et des manipulations des ROBDD, on peut penser cette voie prometteuse dans le cas de formules structurées, ou tout du moins intéressante dans une poursuite naturelle du travail mené dans ce stage.

Remerciements

Je tiens à remercier toute l'équipe Intelligence Artificielle et Système d'Inférences du LRI pour m'avoir aidé dans ce travail, ainsi que J.-F Héry et J.-C. Laleuf de ÉdF pour avoir soutenu l'idée d'un stage sur ce sujet.
Je remercie plus particulièrement Philippe Chatalic pour l'enthousiasme, l'"ecoute et les conseils qu'il m'a offert pendant toute la durée du stage, ainsi que Marie-Christine Rousset pour les remarques qu'elle a pu faire sur les premières versions de ce rapport. Je n'oublie pas non plus Antoine Rauzy qui m'a aimablement envoyé les benchmarks d'*Aralia* et qui a contribué, par l'envoi de plusierus de ses articles, à l'élaboration des idées présentées dans le dernier chapitre.

Table des figures

Liste des tableaux

Bibliographie

[1] H.R. Andersen. An introduction to binary decision diagrams. Course notes.

[2] H. Kautz B. Selman and D. McAllester. Ten challenges in propositional reasoning and search. In *IJCAI-97*, 1997.

[3] R.E. Bryant. Ordered binary-decision diagrams. *ACM Computing Surveys*, 1992.

[4] C. L. Chang and R. C. Lee. *Symbolic Logic and Mechanical Theorem Proving.* Computer Science Classics, Academic Press, 1973.

[5] O. Coudert. Two-level logic minimization: an overview. *Integration, the VLSI journal*, 1994.

[6] B. Selman D. Mitchell and H. Levesque. Hard and easy distributions of sat problems. *AAAI*, 1990.

[7] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 1960.

[8] Johan de Kleer. An improved incremental algorithm for generating prime implicates. In *AAAI*, 1992.

[9] R. Dechter and I. Rish. Directional resolution: The davis-putnam procedure, revisited. In *Proceedings of KR-94*, pages 134–145, 1994.

[10] R. Dechter and I. Rish. To guess or to think? hybrid algorithms for sat. Disponible à www.., 1996.

[11] Yves Dutuit and Antoine Rauzy. Exact and truncated computations of prime implicants of coherent and non-coherent fault trees within aralia. Papier soumis à publication chez Elsevier, 1996.

[12] Mourad Ykhlef Eric Delvalet, Philippe Chatalic. Calcul du résolvant en logique des propositions. un outil pour la validation de système. Rapport Numéro 2 du contrat EDF-LRI (référence: I211J7468).

[13] O. Coudert et J-C. Madre. Une approche intentionnelle du calcul des implicants premiers et essentiels des fonctions booléennes. *Informatique théorique et Applications*, 1994.

[14] J.-F. Héry and J.-C. Laleuf. Logique et modélisation : compatibilité entre données et modèles en logique des propositions. Technical report, EDF, Département Traitement de l'Information et Études Mathématiques, 1995.

[15] J.-F. Héry and J.-C. Laleuf. Logique et modélisation : logique des propositions. Technical report, EDF, Département Traitement de l'Information et Études Mathématiques, 95.

[16] D. Johnson. A theorician's guide to the experimental analysis of algorithms. Disponible sur http://www.research.att.com/ dsj/paper/exper.ps.

[17] Alex Kean and George Tsiknis. An incremental method for generating prime implicants/implicates. *J. Symbolic Computation*, 1990.

[18] G. Logeman M. Davis and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 1962.

[19] S. Minato. Zero-suppressed bdds for set manipulation in combinatorial problems. In *30th ACM/IEEE Design Automation Conference*, 1993.

[20] Antoine Rauzy. An introduction to binary decision diagrams and some of their applications to risk assessment. In *Modélisation et Vérification de Processus Parallèle, MOVEP'96*, 1996.

[21] Antoine Rauzy. Boolean models for reliability analysis: a benchmark. Technical report, LaBRI - URA CNRS 1304 - Université Bordeaux I, 1997.

[22] R. Schrag and M. Crawford. Implicates and prime implicates in random 3-sat. *Artificial Intelligence*, 1996.

[23] Siegel. *Représentation et utilisation de la connaissance en calcul propositionnel*. PhD thesis, Université d'Aix-Marseille II, 1987.

[24] R. M. Sinnamon and J. D. Andrews. Fault tree analysis and binary decision diagrams. In *Proceedings of Annual Reliability and Maintainability Symposium*, 1996.

[25] I. Wegener. Efficient data structures for boolean functions. *Discrete Mathematics*, 1994.

[26] H. Zhang and M. Stickel. Implementing the davis-putnam algorithm by tries. Technical report, University of Iowa, Dept. of Computer Science, 1994.

Annexe A

Démonstrations des théorèmes

A.1 Théorème 2 (*Résolvant et PIs*)

Soit f une formule booléenne mise sous FNC construite sur les variables V, et $f_{V'}$ le résolvant de f sur le sous-ensemble V' des variables de V. Alors pour toute affectation A' des variables de V' satisfaisant $f_{V'}$, il existe au moins une affectation A des variables $(V \setminus V')$ telle que l'affectation $A \cup A'$ satisfasse f. De plus, la restriction de tout modèle de f aux variables V' est un modèle de $f_{V'}$.

Preuve : Soit C_I l'ensemble des clauses impliqués premiers par f. Démontrons par récurrence sur les variables à éliminer l'équivalence entre le résolvant de f et les impliqués premiers de f dont les clauses contenant la variable à éliminer ont été enlevées.
Pour cela, il suffit de montrer que le résolvant de C_I sur une variable x_k est équivalent à C_I dans lequel on a ôté les clauses contenant la variable x_k. Le résolvant se calcule en produisant toutes les clauses C_p issues de la coupure sur x_k. Or, ces clauses sont toutes logiquement impliquées par la formule initiale. Donc, par application de la définition des impliqués premiers, il existe pour chaque clause de C_p une clause de C_I qui l'implique logiquement. Comme d'autre part aucune des clauses de C_p ne contient la variable x_k sous forme positive ou négative, aucune des clauses de C_I impliquant une clause de C_p ne peut contenir la variable x_k. Donc, en notant C'_I l'ensemble des clauses ne contenant pas la variable x_k, on peut écrire que

$$\forall c_p \in C_p, \exists c_I \in C'_I \textit{ tel que } c_I \rightarrow c_p$$

Alors le calcul de résolvant sur C_I se résume à l'élimination des clauses de C_I contenant la variable x_k (l'ensemble C'_I), parce que toute clause produite par coupure sur x_k est déjà impliquée par une clause de C'_I.
Or, par application du théorème de prolongement, les résolvants de deux formules équivalentes sont équivalents logiquement (les deux formules initiales ont les mêmes modèles, donc les deux résolvants ont aussi exactement les mêmes modèles).
Donc, le résolvant de f sur x_k est la conjonction des clauses de C'_I. Il suffit ensuite de noter que les clauses de C'_I sont les impliqués premiers du résolvant de f sur x_k pour pouvoir appliquer la récursivité sur les variables à éliminer.
Au total, l'application des éliminations successives des clauses de C_I contenant les variables sur lesquelles on calcule le résolvant conduit à l'expression du théorème.

A.2 Théorème 5 (*Distribution*)

L'ensemble $Distribue(g, h)$ des clauses composant la distribution des deux ensembles de clauses EC_g et EC_h dont les fonctions caractéristiques sont respectivement g et h est récursivement décrit par :

$$
\begin{aligned}
Distribue(0,0) =& \emptyset \\
Distribue(1,1) =& \{\epsilon\} \\
Distribue(1,0) =& \{\epsilon\} \\
Distribue(0,1) =& \{\epsilon\} \\
Distribue(g,h) =& Distribue(\ (g \wedge \overline{O_k}),\ (h \wedge \overline{O_k}))\ \cup \\
& \{\overline{x_k}\} \times Distribue(\ g \wedge O_k \wedge \overline{S_k},\ h \wedge O_k \wedge \overline{S_k})\ \cup \\
& \{\overline{x_k}\} \times Distribue(\ (g \wedge \overline{O_k}),\ h \wedge O_k \wedge \overline{S_k})\ \cup \\
& \{\overline{x_k}\} \times Distribue(\ g \wedge O_k \wedge \overline{S_k},\ (h \wedge \overline{O_k}))\ \cup \\
& \{x_k\} \times Distribue(\ g \wedge O_k \wedge S_k,\ h \wedge O_k \wedge S_k)\ \cup \\
& \{x_k\} \times Distribue(\ (g \wedge \overline{O_k}),\ h \wedge O_k \wedge S_k)\ \cup \\
& \{x_k\} \times Distribue(\ g \wedge O_k \wedge S_k,\ (h \wedge \overline{O_k}))
\end{aligned}
$$

Preuve : Soit f_1 (resp. f_2) la fonction caractéristique de l'ensemble C_1 (resp. C_2) des clauses composant la formule g (resp. h). Alors la distribution des clauses de C_1 et de C_2 peut s'exprimer récursivement sur les variables x_k. Il suffit d'exprimer la distribution des deux ensembles de clauses en 3 sous ensembles : l'ensemble des clauses ne contenant pas la variable x_k, celui contenant le littéral x_k et celui contenant le littéral $\neg x_k$. Pour cela, on pose C_i^+, C_i^-, C_i' les clauses de $C_{i\in\{1,2\}}$ contenant respectivement x_k, $\neg x_k$ et ni x_k ni $\neg x_k$, alors la distribution de C_1 sur C_2 s'exprime :

$$Distri(C_1, C_2) = (C_1^+ \wedge C_1^- \wedge C_1') \vee (C_2^+ \wedge C_2^- \wedge C_2')$$

soit, en distribuant les connecteurs $\vee$ et $\wedge$

$$
\begin{aligned}
Distri(C_1, C_2) =& (C_1^+ \vee C_2^+) \wedge (C_1^+ \vee C_2^-) \wedge (C_1^+ \vee C_2') \wedge \\
& (C_1^- \vee C_2^+) \wedge (C_1^- \vee C_2^-) \wedge (C_1^- \vee C_2') \wedge \\
& (C_1' \vee C_2^+) \wedge (C_1' \vee C_2^-) \wedge (C_1' \vee C_2')
\end{aligned}
$$

Cette expression se simplifie en éliminant toutes les tautologies :

$$
\begin{aligned}
Distri(C_1, C_2) =& (C_1^+ \vee C_2^+) \wedge (C_1^+ \vee C_2') \wedge \\
& (C_1^- \vee C_2^-) \wedge (C_1^- \vee C_2') \wedge \\
& (C_1' \vee C_2^+) \wedge (C_1' \vee C_2^-) \wedge (C_1' \vee C_2')
\end{aligned}
$$

ou, pour mettre en évidence la présence de la variable dans la formule :

$$
\begin{aligned}
Distri(C_1, C_2) =& (x_k \vee ((C_1^+ \vee C_2^+) \wedge (C_1^+ \vee C_2') \wedge (C_1' \vee C_2^+))) \wedge \\
& (\neg x_k \vee ((C_1^- \vee C_2^-) \wedge (C_1^- \vee C_2') \wedge (C_1' \vee C_2^-))) \wedge \\
& (C_1' \vee C_2')
\end{aligned}
$$

On exprime alors la distribution de C_1 sur C_2 comme :

- L'ensemble des clauses composant la distribution des clauses de C_1 ne contenant ni x_k ni $\neg x_k$ avec les clauses de C_2 ne contenant ni x_k ni $\neg x_k$ (l'ensemble des clauses produites par coupure ne contenant pas la variable x_k),

- Union la disjonction de x_k avec la conjonction de : l'ensemble des clauses de C_1 contenant x_k distribué avec l'ensemble des clauses de C_2 contenant x_k (le littéral x_k apparaît alors dans toutes ces clauses),

- Union la disjonction de x_k avec la conjonction de : l'ensemble des clauses de C_1 contenant x_k distribué avec l'ensemble des clauses de C_2 ne contenant ni x_k ni $\neg x_k$,

- Union la disjonction de x_k avec la conjonction de : l'ensemble des clauses de C_1 ne contenant ni x_k ni $\neg x_k$ distribué avec l'ensemble des clauses de C_2 contenant x_k,

– Union la disjonction de $\neg x_k$ avec la conjonction de : l'ensemble des clauses de C_1 contenant $\neg x_k$ distribué avec l'ensemble des clauses de C_2 contenant $\neg x_k$ (le littéral $\neg x_k$ apparaît alors dans toutes ces clauses),

– Union la disjonction de $\neg x_k$ avec la conjonction de : l'ensemble des clauses de C_1 contenant $\neg x_k$ distribué avec l'ensemble des clauses de C_2 ne contenant ni x_k ni $\neg x_k$,

– Union la disjonction de $\neg x_k$ avec la conjonction de : l'ensemble des clauses de C_1 ne contenant ni x_k ni $\neg x_k$ distribué avec l'ensemble des clauses de C_2 contenant $\neg x_k$

En fait, dans cette énumération, seules les distributions des clauses contenant deux littéraux opposés sont écartées.

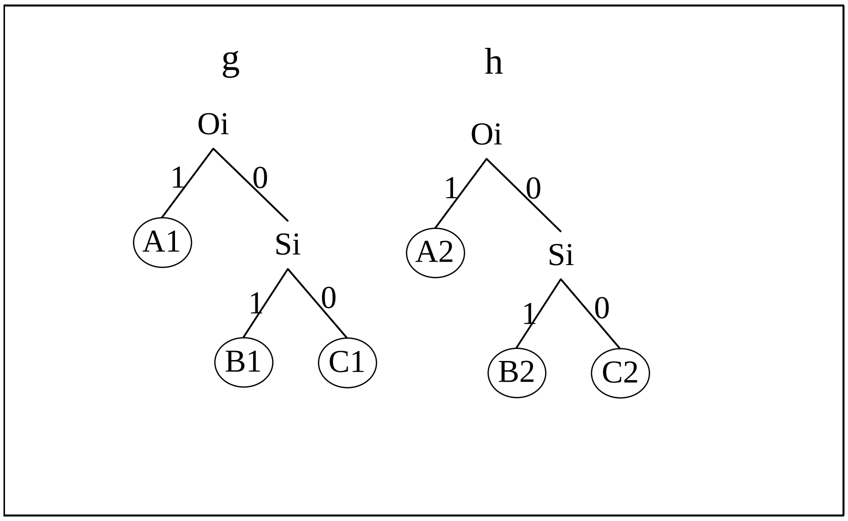

FIG. A.1 - *Représentation des deux ROBDD à distribuer sur x_i*

Sur la figure A.1, qui représente deux ROBDD à distribuer sur la variable x_i, rappelons que l'ensemble des clauses de g est alors :

– L'ensemble des clauses de $A1$

– Union l'ensemble des clauses représentant la disjonction de x_k sur la conjonction de l'ensemble des clauses de $B1$

– Union l'ensemble des clauses représentant la disjonction de $\neg x_k$ sur la conjonction de l'ensemble des clauses de $C1$

la distribution des deux ROBDD peut alors s'exprimer, en confondant les clauses avec les ensembles de littéraux les composant :

– $Distribution(A1, A2)$

– $\cup\{x_i\} \times Distribution(B1, B2)$

– $\cup\{x_i\} \times Distribution(B1, A2)$

– $\cup\{x_i\} \times Distribution(A1, B2)$

– $\cup\{\neg x_i\} \times Distribution(C1, C2)$

– $\cup\{\neg x_i\} \times Distribution(C1, A2)$

– $\cup\{\neg x_i\} \times Distribution(A1, C2)$

On peut alors exprimer la distribution sur les ROBDD codant les deux fonctions caractéristiques pour obtenir la formule que l'on voulait démontrer.

A.3 Théorème 6 (*Distribution par opération logique*)

$[Distri_1(A,B)](c)$ renvoie vrai si et seulement si le MétaProduit de la clause c fait partie de la distribution des deux ensembles de clauses respectifs A et B.

Preuve : On procède en fait d'une manière similaire à la démonstration du théorème précédent. On impose récursivement des conditions sur les deux ensembles de clauses distribuées en fonction des littéraux présents dans la clause.

On a écrit le principe de récurrence de la définition de $Distri_i(A,B)$ dans le rapport, page 51. On peut donc dire qu'une clause est la fin au rang i d'une clause distribuée sur A et B si et seulement si, pour $i < n$:

- x_i fait partie de la clause ET la fin au rang $i+1$ de cette clause est la distribution de deux clauses sur les ensembles $A_{Contenant\ x_i}$ et $B_{Ne\ Contenant\ pas\ \overline{x_i}}$, OU
- x_i fait partie de la clause ET la fin au rang $i+1$ de cette clause est la distribution de deux clauses sur les ensembles $A_{Ne\ Contenant\ pas\ \overline{x_i}}$ et $B_{Contenant\ x_i}$, OU
- $\neg x_i$ fait partie de la clause ET la fin au rang $i+1$ de cette clause est la distribution de deux clauses sur les ensembles $A_{Contenant\ \overline{x_i}}$ et $B_{Ne\ Contenant\ pas\ x_i}$, OU
- $\neg x_i$ fait partie de la clause ET la fin au rang $i+1$ de cette clause est la distribution de deux clauses sur les ensembles $A_{Ne\ Contenant\ pas\ x_i}$ et $B_{Contenant\ \overline{x_i}}$, OU
- la variable x_i n'est pas dans la clause ET la fin au rang $i+1$ de cette clause est la distribution de deux clauses sur les ensembles $A_{Ne\ Contenant\ pas\ x_i\ ni\ \overline{x_i}}$ et $B_{Ne\ Contenant\ ni\ x_i\ ni\ \overline{x_i}}$

Et pour $i = n$

- x_n fait partie de la clause ET aucun des ensembles $A_{Contenant\ x_n}$ et $B_{Ne\ Contenant\ Pas\ \overline{x_n}}$ est vide OU
- x_n fait partie de la clause ET aucun des ensembles $A_{Ne\ Contenant\ Pas\ \overline{x_n}}$ et $B_{Contenant\ x_n}$ est vide, OU
- $\neg x_n$ fait partie de la clause ET aucun des ensembles $A_{Contenant\ \overline{x_n}}$ et $B_{Ne\ Contenant\ Pas\ x_n}$ est vide, OU
- $\neg x_n$ fait partie de la clause ET aucun des ensembles $A_{Ne\ Contenant\ Pas\ x_n}$ et $B_{Contenant\ \overline{x_n}}$ est vide, OU '
- la variable x_n n'est pas dans la clause ET les ensembles $A_{Ne\ Contenant\ Pas;x_n\ ni\ \overline{x_n}}$ et $B_{Ne\ Contenant\ Pas;x_n\ ni\ \overline{x_n}}$ ne sont pas vides

La formule n'est que la formalisation logique du principe récursif décrit. Donc $[Distri_1(A,B)](c)$ renvoie vrai si et seulement si la fin au rang 1 de c est une clause de la distribution de A sur B, ce qui signifie bien que c est une clause de la distribution de A sur B.