Chapitre 2

Satisfaisabilité Propositionnelle (SAT) et Modulo Théories (SMT)

Sylvain Conchon
Laurent Simon

Le cours est en deux parties, la première étant axée sur les solveurs SAT (satisfaisabilité propositionnelle), la seconde sur les solveurs SMT (satisfaisabilité modulo théories).

Nous présenterons d'abord les éléments clés de ce qui compose les "Solveurs SAT Modernes" pour la résolution du problème de la satisfaisabilité propositionnelle. Après une introduction historique sur la quête de la résolution pratique du problème NP-Complet canonique "SAT", nous expliquerons les caractéristiques algorithmiques et les structures de données des solveurs CDCL (Conflict Driven Clause Learning), les points essentiels devant être intégrés à ce solveurs.

À l'instar des solveurs SAT, les solveurs SMT sont de plus en plus utilisés, dans divers domaines. Bénéficiant des progrès extraordinaires réalisés ces dernières années par les solveurs SAT sur lesquels ils sont basés, les démonstrateurs SMT permettent de décider de la satisfaisabilité de formules logiques contenant des symboles de théories particulières. Nous tenterons dans cette seconde partie de faire découvrir les principaux concepts sous-jacents à la conception et l'implémentation de tels solveurs, plus particulièrement les théories utilisées en pratique et leurs procédures de décision ainsi que les techniques de combinaison de théories.

2.1 Introduction

À la croisée des mathématiques discrètes et de l'informatique théorique se trouve l'un des problèmes les plus importants de l'informatique : le problème SAT, pour satisfaisabilité (ou satisfiabilité). Intuitivement, ce problème capture la difficulté de toute une famille de problèmes *difficiles* que l'on rencontre dans un très grand nombre de problèmes théoriques ou pratiques. Le problème SAT tient une place à part dans cette famille de problèmes difficiles. Cela est tout d'abord lié à l'histoire, puisque SAT a été le premier problème *difficile*, au sens de la théorie de la complexité. Mais ce qui le rend encore aujourd'hui essentiel à un très grand nombre d'applications, ce sont les progrès actuels observés dans sa résolution pratique. Comme nous allons le voir, malgré une impossibilité théorique forte, les solveurs actuels arrivent à résoudre des problèmes de taille gigantesque.

Le problème SAT s'exprime en logique propositionnelle. Extrêmement simple, cette logique colle parfaitement à la force de calcul brute des machines actuelles. Son apparente simplicité permet en effet l'élaboration d'algorithmes compacts et efficaces – en pratique – pour attaquer toute une classe de problèmes importants, mais intraitables – en théorie –. Il est frappant de constater combien cette logique est ancienne, puisqu'elle a été formalisée par Aristote lui-même. C'est donc de manière assez paradoxale que l'on retrouve cette logique au cœur de démonstrateurs automatiques permettant la vérification de microprocesseurs, de programmes, ou encore de problèmes de cryptographie, de théorèmes mathématiques et bioinformatiques de première importance. Cela explique certainement pourquoi Edmund Clarke, l'un des Turing Awards 2007, ait ainsi déclaré « la résolution pratique du problème SAT est une technologie clé pour l'informatique du 21ème siècle » [25].

Dans la section suivante, nous proposons tout d'abord d'introduire le type de problèmes typiquement accessibles à ce formalisme, puis nous présentons l'historique des progrès observés dans la résolution pratique du test de satisfaisabilité (SAT), en nous focalisant sur les techniques actuelles utilisées en pratique.

Le problème SMT, pour satisfaisabilité modulo théories, est une généralisation du problème SAT à des logiques plus riches, étendues avec des symboles de fonction et de prédicat particuliers, dont le sens est fixé par des théories prédéfinies. Ainsi, on peut s'intéresser au problème de la satisfaisabilité modulo la théorie de l'arithmétique linéaire sur les entiers (LIA), la théorie des tableaux (Array) ou la théorie des vecteurs de bits (BV), etc. Plus généralement, la théorie sous-jacente à un problème SMT

est souvent une union de théories plus élémentaires (ainsi, on s'intéressera aux problèmes SMT LIA+BV, ou LIA+Array, etc.).

Nous verrons dans la deuxième partie de ce cours que la résolution d'un problème SMT s'appuie sur la coopération efficace entre un solveur SAT et des petits moteurs de preuve pour les théories, appelés *procédures de décision*. Après avoir présenté deux exemples de théories (la théorie de l'égalité et la théorie des inéquations), nous verrons comment interfacer un solveur SAT avec une procédure de décision. Nous terminerons cette partie par une présentation des techniques pour combiner les procédures de décision de différentes théories.

2.2 Problème de satisfaisabilité en logique propositionnelle

Dans cette section, nous présentons la logique propositionnelle depuis sa définition syntaxique jusqu'à l'algorithmique la plus récente permettant de résoudre en pratique des instances de taille industrielle. Nous présentons également l'intérêt théorique du problème SAT et présentons les avancées de manière chronologique.

2.2.1 Logique propositionnelle et SAT : définitions

Formalisée il y a plus de 2000 ans, cette logique remonte à la naissance du mot « raisonnement » lui-même, le mot « syllogisme ». En Grec ancien, un syllogisme portait trois sens : calcul, hypothèse et raisonnement. Ce *raisonnement-calcul* se caractérise par l'articulation entre des propositions (les *prémisses*) et une *conclusion*, qui déjà, pour Aristote, ne pouvaient prendre comme valeurs que vrai ou faux. Parmi les syllogismes proposés, le *modus ponens* est certainement l'un des plus simples et des plus connus. Il exprime simplement que « *si* a *est vrai* **et** *si* a *implique* b **alors** b *est vrai* ».

Une logique se définit d'abord de manière syntaxique, la sémantique étant définie par la suite sur ce langage. Intuitivement, les machines ne travailleront qu'au niveau syntaxique, manipulant des expressions vides de sens. Ainsi, le langage des formules propositionnelles peut se définir formellement à partir d'un ensemble fini de variables propositionnelles (ou symboles propositionnels) $\mathcal{V}$, de deux constantes *vrai* et *faux* ainsi que d'un ensemble de *connecteurs logiques* : $\neg$ (négation), $\vee$ (disjonction), $\wedge$ (conjonction), $\rightarrow$ (implication), $\leftrightarrow$ (équivalence), $\oplus$ (exclusion). On note $Var(f)$ l'ensemble des variables propositionnelles de $\mathcal{V}$ apparaissant dans la formule f, et on nomme les formules de base ainsi : un **littéral** (ou

x	y	$x \vee y$	$x \wedge y$	$x \rightarrow y$	$x \leftrightarrow y$	$x \oplus y$	$\neg x$
faux	*faux*	*faux*	*faux*	*vrai*	*vrai*	*faux*	*vrai*
faux	*vrai*	*vrai*	*faux*	*vrai*	*faux*	*vrai*	*vrai*
vrai	*faux*	*vrai*	*faux*	*faux*	*faux*	*vrai*	*faux*
vrai	*vrai*	*vrai*	*vrai*	*vrai*	*vrai*	*faux*	*faux*

TABLE 2.1 – *Valuation sur les fonctions usuelles définies sur les connecteurs* $\vee, \wedge, \rightarrow, \leftrightarrow, \oplus$ *et* $\neg$.

formule atomique) est une formule de la forme x (littéral positif) ou $\neg x$ (littéral négatif) avec $x \in \mathcal{V}$. Les littéraux x et $\neg x$ sont dits **littéraux complémentaires**. Une **clause** est une disjonction finie de littéraux (par exemple $C = x_1 \vee \neg x_2 \vee x_3$ est une clause). Un **produit** est une conjonction finie de littéraux (par exemple $P = x_1 \wedge x_2 \wedge \neg x_3$ est un produit).

La sémantique classique de la logique propositionnelle repose sur la notion *d'interprétation*. On se donne un ensemble $\mathbb{B} = \{\mathtt{V}, \mathtt{F}\}$, constitué de deux *valeurs de vérité*, et l'on associe chaque variable propositionnelle à une de ces valeurs. Une **interprétation** I d'un ensemble de variables $V \subseteq \mathcal{V}$ est une application ayant pour domaine V et pour co-domaine $\mathbb{B} = \{\mathtt{V}, \mathtt{F}\}$. Lorsque $V \neq \mathcal{V}$, on dit que l'interprétation est *partielle* (au contraire d'une interprétation *totale*).

Plus simplement, une interprétation totale donne à chaque variable une valeur de vérité unique qui permettra d'évaluer la formule propositionnelle. Souvent, on peut calculer la valeur de la formule sans avoir à donner à chaque variable une valeur (par exemple $x \vee y$ est vrai dès que x est vrai, sans avoir à donner de valeur à y). On parle donc dès lors d'interprétation partielle.

Par exemple, la formule $(x \vee \neg y) \wedge (\neg x \vee \neg z)$ est évaluée à vrai selon l'interprétation partielle $x = \mathtt{V}, z = \mathtt{F}$ et fausse suivant l'interprétation partielle $x = \mathtt{F}, y = \mathtt{V}, z = \mathtt{V}$. Plus formellement, pour évaluer une formule étant donnée une interprétation des variables, on peut récursivement définir l'interprétation de la formule en utilisant la table de vérité classique utilisée sur les connecteurs usuels de la logique (table 2.1).

SAT : à la recherche de modèles

Une fois que la sémantique est fixée, on peut se demander si, étant donnée une formule f, il existe une interprétation I qui la **satisfait** (on aurait $[\![f]\!]_I = \mathtt{V}$). Si c'est le cas, on dit que I est un **modèle** de f, ce que l'on note $I \models f$. Inversement, si $[\![f]\!]_I = \mathtt{F}$, on dit que I **falsifie** f et que I est un

contre-modèle de f, ce qui se note $I \not\models f$. Le problème qui nous intéresse ici est donc de savoir si une formule donnée admet un modèle (problème de **satisfaisabilité**, SAT). Intuitivement, on voit bien que si aucun indice ne nous aide, on pourrait avoir à essayer l'un après l'autre les $2^{Var(f)}$ modèles, ce qui est absolument impossible en pratique (cet ordre de grandeur est uniquement indicatif : calculer la complexité la plus faible d'un algorithme résolvant le problème SAT est cependant assez étudié, voir [201] pour un état de l'art, les résultats donnés ayant un majorant de l'ordre de $1.48^{Var(f)}$). À titre d'exemple, si l'on devait énumérer tous les modèles possibles sur une formule à 260 variables, alors les 2^{260} modèles à tester dépasseraient l'une des estimations communes du nombre de particules de l'univers. Quand on sait, de plus, que les solveurs modernes traitent des formules avec plusieurs millions de variables en quelques minutes, on comprend combien le problème de la résolution pratique de SAT est fascinant. Le nombre de modèles potentiels peut rapidement dépasser l'entendement.

Lorsqu'une formule n'admet aucun modèle, on dit qu'elle est **insatisfaisable** (on dit aussi que f est une **contradiction** ou est **incohérente** ou UNSAT), ce qui est alors noté $\models \neg f$. Au contraire, si toute interprétation sur $Var(f)$ est un modèle de f, alors on dit que f est une **tautologie**, ce qui est noté $\models f$ (on parle aussi de formule **valide**). Par exemple : $(x \rightarrow (y \rightarrow x))$ est une tautologie ; $(x \leftrightarrow (\neg x))$ est une formule insatisfaisable ; et $(x \vee (y \oplus z))$ est une formule satisfaisable (toute interprétation satisfaisant x est un modèle), mais non tautologique (l'interprétation affectant x à **F** et y, z à **V** est un contre-modèle).

La règle de résolution Avant de pouvoir utiliser en pratique une règle syntaxique (adaptée à l'automatisme des machines) permettant de raisonner sur une formule donnée, il faut pouvoir manipuler la formule et l'écrire sous une forme canonique, adapté à la règle de *résolution* que l'on va introduire dans la suite.

Une formule g est une **conséquence sémantique** d'une formule f (noté $f \models g$), si tout modèle de f est un modèle de g. De plus, pour vérifier si g est une conséquence logique de f, il « suffit » de vérifier que la formule $f \wedge (\neg g)$ est bien UNSAT. Lorsque la conséquence est vérifiée dans les deux directions (*i.e.* si $f \models g$ et $g \models f$), alors les deux formules f et g ont exactement les mêmes modèles, et on dit qu'elles sont **équivalentes**. Cette notion permet certaines transformations syntaxiques des formules, dès lors que celles-ci préservent l'équivalence sémantique. On pourra par exemple utiliser l'*idempotence* du connecteur $\vee$ pour réécrire $(f \vee f)$ en f. De nombreuses lois classiques de la logique permettent de réécrire une formule, comme l'élimination de la double négation ($f \equiv \neg(\neg f)$) ou encore

les *lois de De Morgan*. Citons tout de même deux dernières règles à titre d'illustration. La distributivité de $\vee$ sur $\wedge$: $(f \vee (g \wedge h)) \equiv (f \wedge g) \vee (f \wedge h)$; et l'élimination de $\oplus$: $(f \oplus g) \equiv (\neg f \wedge g) \vee (f \wedge \neg g)$.

Les formes normales Le problème SAT, s'il est défini sur n'importe quelles formules en logique propositionnelle, s'exprime généralement sur des formules écrites sous **forme normale conjonctive** ($\mathcal{FNC}$) (conjonction de clauses). Une formule f est dite sous **forme normale disjonctive** ($\mathcal{FND}$) si et seulement si elle correspond à une disjonction de produits. On notera que le problème SAT est trivial sur une formule sous $\mathcal{FND}$.

En profitant des lois usuelles des connecteurs logiques, (comme les *lois de De Morgan*, la distributivité, l'associativité, ...), on peut maintenant écrire toute formule f en une formule équivalente, dans laquelle notamment tous les connecteurs $\rightarrow$, $\leftrightarrow$ et $\oplus$ ont été éliminés, ainsi que les doubles négations. Il ne reste alors plus qu'une formule bâtie sur la disjonction, la conjonction et la négation. Si on prend soin de « pousser » les négations jusqu'aux variables, on obtient une formule sous *forme normale négative* ($\mathcal{FNN}$). On peut en dernier lieu utiliser les propriétés de distributivité des opérateurs pour écrire la formule de départ sous $\mathcal{FNC}$ ou $\mathcal{FND}$, au choix.

Par exemple, si on doit travailler sur la formule $\neg(f \oplus g)$, on va d'abord utiliser la règle vue pour la réécrire en $\neg((\neg f \wedge g) \vee (f \wedge \neg g))$ pour ensuite pousser la négation la plus extérieure jusqu'aux variables : $(f \vee \neg g) \wedge (\neg f \vee g)$. On obtient dès lors une $\mathcal{FNN}$.

Malheureusement en pratique, ces substitutions ne suffisent pas pour transformer n'importe quelle formule sous $\mathcal{FNC}$, car la taille de la formule normalisée peut croître de manière exponentielle si l'on se restreint aux méthodes de réécritures ci-dessus.

Tseitin [193] a introduit un mécanisme permettant de normaliser n'importe quelle formule sous $\mathcal{FNC}$ en préservant non pas l'équivalence, mais l'**équi-satisfiabilité**. Intuitivement, il s'agit d'introduire un ensemble de nouvelles variables dont la valeur de vérité représente la satisfaisabilité, ou non, de sous-formules. Par exemple, lorsqu'il faut encoder un *circuit*, une variable x est associée à chaque *porte* $f(x_1, \ldots, x_n)$ telle que x soit sémantiquement équivalente à la valeur f de la porte : $x \leftrightarrow f(x_1, \ldots, x_n)$. En répétant ce processus au niveau de chaque sous formule élémentaire, la formule globale peut s'écrire ensuite simplement comme la conjonction de tous les encodages, ce qui évite l'explosion due à la distribution des connecteurs logiques les uns sur les autres. Par exemple, si la sous-formule f s'écrit comme la disjonction $g \vee h$, on introduit deux nouvelles variables x_g et x_h représentant la satisfaisabilité des deux sous-formules g et h, respectivement. La satisfaisabilité de f peut alors s'écrire comme

la $\mathcal{FNC}$ suivante $(\neg f \vee x_g \vee x_h) \wedge (f \vee \neg x_g) \wedge (f \vee \neg x_h)$ (ce qui encode l'équivalence logique introduite). En pratique, l'introduction de nouvelles variables est essentielle pour contenir l'explosion combinatoire (syntaxique) que représenterait l'écriture de toute formule sous $\mathcal{FNC}$.

La règle de résolution

La résolution [166] est l'une des règles de déduction de base en logique propositionnelle : soient deux clauses c_1, c_2 et x une variable propositionnelle, la règle de résolution est la règle d'inférence suivante :

$$(x \vee c_1),\ (\neg x \vee c_2) \vdash_{\mathcal{R}}\ c_1 \vee c_2$$

La clause $c_1 \vee c_2$ est appelée **clause résolvante** sur x des clauses $(x \vee c_1)$ et $(\neg x \vee c_2)$. Cette règle est très proche du raisonnement commun. Elle peut être vue comme une application directe de la règle de *coupure* (si on a $f \rightarrow g$ et $g \rightarrow h$, alors on a $f \rightarrow h$), dans le cas particulier où f est une clause, g un littéral et h un produit. Bien entendu, cette simple règle de déduction est correcte : on a bien $(c_1 \vee x) \wedge (c_2 \vee \neg x) \models c_1 \vee c_2$. Elle est aussi complète pour la réfutation : si f est insatisfaisable, alors on a la garantie qu'il existe bien une suite finie de résolutions permettant de déduire la clause vide.

Une preuve par résolution dans Σ d'une clause c' fondamentale est une suite de clauses $P = c_1, \ldots, c_k$ telle que $c_k = c'$ et telle que l'on ait, pour tout $i \leqslant k$: ou bien $c_i \in \Sigma$; ou bien il existe c_m et c_n ($m < i$ et $n < i$) dans P tels que c_i soit la résolvante de c_m et c_n. On note $\Sigma \vdash_{\mathcal{R}} c'$ le fait qu'il existe une telle preuve. Sauf mention contraire, nous notons simplement $\Sigma \vdash c'$ pour $\Sigma \vdash_{\mathcal{R}} c'$.

2.2.2 Difficulté théorique de SAT, et problèmes pratiques

On ne peut parler du problème SAT sans aborder quelques notions de complexité. Il s'agit en effet du premier problème démontré comme NP-complet [51]. Cette notion est primordiale à bien des égards, notamment parce qu'elle permet de classer les problèmes selon une complexité indépendante des algorithmes qui les résolvent en pratique. Pour une introduction à ce sujet, le lecteur intéressé pourra consulter les incontournables [82] et [113].

Pour simplifier, nous introduisons donc la notion de classe de complexité à l'aide de la notion de fonction calculable. Intuitivement, une fonction f est dite *calculable* simplement s'il existe un algorithme qui permet de calculer son résultat (nous imposons juste à ce niveau que le programme finisse un jour son calcul, quel que soit le résultat de la fonction). On peut

alors mesurer le temps de calcul de f par le nombre d'étapes élémentaires de l'algorithme.

La classe P est l'ensemble des problèmes de décision pouvant être résolus en temps polynomial. En terme de fonction calculable, ces problèmes sont caractérisés par la réponse à la question « Que vaut $f(x)$? », avec f calculable en temps polynomial par rapport à la taille de la donnée x. L'ensemble des problèmes appartenant à cette classe sont parfois appelés *faciles*. En théorie, le passage à l'échelle des exemples traités est toujours possible, moyennant des machines plus puissantes, ce qui n'est plus le cas pour les problèmes appartenant aux classes définies ci-dessous (sous l'hypothèse $P \neq NP$).

La classe NP est l'ensemble des problèmes de décision pouvant être résolus en temps polynomial de manière non déterministe. Cette classe correspond à l'ensemble des problèmes de décision s'exprimant logiquement comme $\exists x.f(x)$, ou « existe-t-il un x tel que l'on ait $f(x)$ évalué à vrai ? », avec f une fonction calculable en temps polynomial. Intuitivement, on voit que le problème consiste à *deviner* la bonne solution x (c'est la partie non déterministe). En pratique, les meilleurs algorithmes connus de recherche de ces solutions sont de coût exponentiel dans le pire des cas. Pourtant, rien ne prouve que l'on ne trouvera jamais d'algorithmes polynomiaux pour résoudre des problèmes NP-complets, même si la conjecture $P \neq NP$ est très forte. Quoi qu'il en soit, ces problèmes ont une forte caractéristique pratique : on peut **vérifier une solution du problème en temps polynomial** (il suffit de calculer $f(x)$).

Étant donné un problème de décision, nous pouvons aussi définir son problème complémentaire, comme $\forall x.f(x)$, ou « a-t-on $f(x)$ pour tout x ? », où, encore une fois, f est une fonction calculable en temps polynomial par rapport à la taille de x. La classe $CoNP$ est l'ensemble de ces problèmes, dont le problème complémentaire appartient à NP. De nouveau, une très forte conjecture, non encore prouvée ou infirmée, découle de cette définition : $NP \neq CoNP$.

Enfin, grâce à la notion de complétude, nous avons en main les armes pour classer les problèmes entre eux. Intuitivement, les problèmes NP-Complets sont les plus difficiles des problèmes de NP : si l'on savait en résoudre, ne serait-ce qu'un seul, en temps polynomial alors on saurait les résoudre tous en temps polynomial (la complétude est basée sur la notion de *réduction polynomiale*, non introduite ici par souci de place).

SAT a donc été prouvé comme NP-complet par le fameux théorème de Cook-Levin. Cela le place au centre de la hiérarchie polynomiale, qui classe les problèmes par difficulté croissante.

Intérêt de SAT, d'un point de vue pratique

Attaquer de front le problème SAT est longtemps resté extrêmement difficile en pratique, voire un problème à fuir à tout prix. Pour étudier la performance pratique des solveurs SAT, il a tout d'abord été proposé de générer des problèmes aléatoires, au début des années 1990 [132]. Leur utilisation dans le but de progresser pour la résolution de problèmes réels n'a cependant duré qu'un temps, jusqu'à ce que soit mis à jour certains phénomènes insoupçonnés de ces instances [133]. À la fin des années 1990, en effet, la séparation entre solveurs dédiés aux formules aléatoires et ceux dédiés aux formules industrielles (ou issus du monde réel) était consommée [115, 116].

Puis, en 2001, conjointement avec l'apparition de ce que l'on appelle les « solveurs modernes », est apparue la notion de *Bounded Model Checking* [24, 155] qui permet, en déroulant un certain nombre d'étapes, de transformer en SAT des problèmes de logiques temporelles, ou encore d'atteignabilité d'automates. Lorsque ces automates représentent le fonctionnement d'un microprocesseur, un état particulier, *faute*, est généralement représenté, et l'existence d'un chemin entre l'état initial et la faute représente un exemple de mauvais fonctionnement. Cette longueur de chemin doit être bornée (*Bounded*) pour garantir la transformation en logique propositionnelle. Il est aussi possible de tester l'équivalence de deux circuits (le circuit de référence et le circuit optimisé, effectivement implanté dans un CPU) grâce à SAT, en construisant une formule dont la satisfiabilité implique l'équivalence des deux circuits. On va trouver aussi des moteurs SAT dans l'état de l'art des méthodes de résolution de problèmes importants de bioinformatique [47], de cryptographie ou encore au cœur des méthodes les plus performantes en planification (une plongée dans les nombreux problèmes soumis aux compétitions SAT [114] permet de voir l'étendue des problèmes pouvant être attaqués par SAT). Le nombre de problèmes pour lesquels SAT est devenu essentiel ne cesse de croître. Ainsi, plus récemment, on peut aussi utiliser les SAT solveurs comme des oracles NP de manière efficace, et concevoir des applications demandant des milliers d'appels SAT.

2.2.3 Résoudre SAT en pratique

Depuis plus de vingt-cinq ans, et la première compétition DIMACS en 1993 [64], une grande partie de la communauté SAT se mobilise pour tenter de résoudre, en pratique, ce problème en théorie difficile. Ces progrès théoriques et pratiques, motivés par des compétitions annuelles [114], sont certainement l'une des évolutions majeures de l'intelligence artificielle

(IA) de ces dernières années (Moshe Vardi, Professeur à la Rice University, USA, propose d'ailleurs d'utiliser le terme de « Deep Solving » pour rendre compte de cette évolution majeure).

Dans ce cours, nous nous limiterons aux algorithmes complets permettant de prouver à la fois SAT et UNSAT. Nous omettons ainsi tout un pan de recherche autour des méthodes incomplètes et méthodes de recherches locales, parfois très efficaces en pratique.

DP60 : Davis et Putnam Dans sa première version [56], l'algorithme de Davis et Putnam fonctionnait par « oublis » successifs des variables de la formule. Prenons la décomposition de Shannon, on sait que toute formule f peut se réécrire en $f' \equiv (x \wedge f|_{\{x\}}) \vee (\neg x \wedge f|_{\{\neg x\}})$ pour tout $x \in Var(f)$, la formule $f|_{\{x\}}$ (resp. $f|_{\{\neg x\}}$) étant f dans laquelle toutes les occurrences de x ont été remplacées par *vrai* (resp. $faux$). La formule obtenue f' n'a plus aucune occurrence de x, mais préserve la satisfaisabilité de f. La difficulté pratique de DP vient de la réécriture de f' sous $\mathcal{FNC}$, celle-ci faisant intervenir la distribution des deux ensembles de clauses obtenus l'un sur l'autre.

Prenons la formule sous $\mathcal{FNC}$ suivante (on utilise ici la notation plus compacte $\overline{x}$ pour $\neg x$) :

$x_1 \vee x_4$
$\overline{x_1} \vee x_4 \vee x_{14}$
$\overline{x_1} \vee \overline{x_3} \vee \overline{x_8}$
$x_1 \vee x_8 \vee x_{12}$
$x_1 \vee x_5 \vee \overline{x_9}$
$x_2 \vee x_{11}$
$\overline{x_3} \vee \overline{x_7} \vee x_{13}$
$\overline{x_3} \vee \overline{x_7} \vee \overline{x_{13}} \vee x_9$
$x_8 \vee \overline{x_7} \vee \overline{x_9}$

Si on veut éliminer x_1, on peut factoriser les clauses contenant x_1 et celles contenant $\overline{x_1}$. On obtient :

$$x_1 \vee \begin{pmatrix} x_4 \\ x_8 \vee x_{12} \\ x_5 \vee \overline{x_9} \end{pmatrix}$$
$$\overline{x_1} \vee \begin{pmatrix} x_4 \vee x_{14} \\ \overline{x_3} \vee \overline{x_8} \end{pmatrix}$$
$x_2 \vee x_{11}$
$\overline{x_3} \vee \overline{x_7} \vee x_{13}$
$\overline{x_3} \vee \overline{x_7} \vee \overline{x_{13}} \vee x_9$
$x_8 \vee \overline{x_7} \vee \overline{x_9}$

Ce qui se traduit, en utilisant la règle de résolution sur x_1 :

$$\begin{pmatrix} x_4 \\ x_8 \vee x_{12} \\ x_5 \vee \overline{x_9} \end{pmatrix} \vee \begin{pmatrix} x_4 \vee x_{14} \\ \overline{x_3} \vee \overline{x_8} \end{pmatrix}$$
$x_2 \vee x_{11}$
$\overline{x_3} \vee \overline{x_7} \vee x_{13}$
$\overline{x_3} \vee \overline{x_7} \vee \overline{x_{13}} \vee x_9$
$x_8 \vee \overline{x_7} \vee \overline{x_9}$

La variable x_1 a bien été éliminée, mais la formule n'est plus sous $\mathcal{FNC}$. Il faut maintenant distribuer les deux ensembles l'un sur l'autre pour obtenir la nouvelle formule suivante :

$x_4 \vee x_{14}$
$x_4 \vee \overline{x_3} \vee \overline{x_8}$
$x_8 \vee x_{12} \vee x_4 \vee x_{14}$
$x_5 \vee \overline{x_9} \vee x_4 \vee x_{14}$
$x_5 \vee \overline{x_9} \vee \overline{x_3} \vee \overline{x_8}$
$x_2 \vee x_{11}$
$\overline{x_3} \vee \overline{x_7} \vee x_{13}$
$\overline{x_3} \vee \overline{x_7} \vee \overline{x_{13}} \vee x_9$
$x_8 \vee \overline{x_7} \vee \overline{x_9}$

Ici, on a bien une $\mathcal{FNC}$ qui est satisfaisable si et seulement si la formule initiale était satisfaisable. De plus, cette formule ne contient plus x_1.

Pour prouver la satisfaisabilité (ou plutôt la non satisfaisabilité) il suffit d'éliminer toutes les variables de la formule. Si l'on n'obtient pas la clause vide (clause vide de tout littéral, obtenue en faisant par exemple une résolution entre la clause x et la clause $\neg x$), alors la formule est satisfaisable. En général, la succession d'élimination de variable va cependant entraîner une explosion combinatoire du nombre de clauses à gérer, ce qui rend cette approche inutilisable en pratique, sauf cas particuliers. En effet, DP, revisité dans [60, 163, 178], a fait l'objet de beaucoup de travaux, de par ses liens avec des classes polynomiales estimées proches d'instances du monde réel (basée sur la *largeur induite* [195]), mais aussi, plus récemment, pour traiter des problèmes au dessus de SAT, notamment des problèmes de *Compilations de Bases de Connaissances* ou QBF. En effet, si l'on met de côté les clauses produites tout au long du calcul, il est possible, lorsque toutes les variables ont été éliminées, d'énumérer tous les modèles de la formule dans un temps proportionnel à leur nombre [60]. On voit que DP répond de fait à un problème plus difficile que le *simple* problème de satisfaisabilité. À part pour quelques problèmes spécifiques de faible largeur induite [60], DP60 n'a jamais vraiment pu rivaliser avec la version de 1962, basée sur un parcours systématique des modèles potentiels de la formule, avec retours arrières en cas d'échecs. La largeur induite est une mesure structurelle des

Algorithme 1 : Procédure DPLL62.

Initialisation : $\mathcal{F}$ une formule propositionnelle;
Procédure DPLL($\mathcal{F}$)
si *il existe dans* $\mathcal{F}$ *un* ***littéral pur*** ℓ **alors** Renvoyer DPLL($\mathcal{F}_\ell$);
si *il existe dans* $\mathcal{F}$ *une* ***clause unitaire*** ℓ **alors** Renvoyer DPLL($\mathcal{F}_\ell$);
si $\mathcal{F}$ *contient au moins une* ***clause vide*** **alors** Renvoyer faux;
si $\mathcal{F}$ ***est vide*** **alors** Renvoyer vrai;
$v \leftarrow$ une variable de $\mathcal{F}$ **(Heuristique de choix pour la division)**
si DPLL($\mathcal{F}_v$) **alors**
  Renvoyer vrai
sinon
  Renvoyer DPLL($\mathcal{F}_{\neg v}$)

problèmes, permettant de borner l'accroissement de la taille de la formule lors de l'élimination successive des variables. On en retrouvera toutefois une version limitée – mais indispensable aux approches *modernes* –, dans les prétraitements des formules, section 2.2.3.

DPLL62 : l'anticipation doit limiter les retours arrières Plutôt que de réécrire la formule sous $\mathcal{FNC}$ après chaque élimination de variable, et de tenter de lutter contre l'explosion combinatoire en mémoire comme dans DP60, il a été proposé dans DPLL62 [55] de remplacer la disjonction dans f' (voir ci-dessus) par une alternative de choix, une *division* de l'espace de recherche, en vérifiant d'abord la satisfaisabilité de $f|_{\{x\}}$ puis, si le résultat est négatif, en vérifiant la satisfaisabilité de $f|_{\{\neg x\}}$. Cette notation (parfois notée plus simplement f_x quand l'ensemble de littéraux est un singleton) représente la simplification de la formule f par l'interprétation partielle indicée. Cette interprétation partielle est notée par l'ensemble des littéraux vrais (plutôt que la valeur de chaque variable). L'algorithme DPLL s'écrit ainsi simplement sous forme de retours arrières chronologiques, comme illustré par l'algorithme 1. Une liste non exhaustive de quelques solveurs DPLL62 marquants est par exemple SATZ [119], KCNFS [66], SATO [203] et GRASP [177].

On notera que la règle du *littéral pur* (une variable n'apparaissant que positivement ou négativement dans la formule) n'est pas toujours incorporée dans les solveurs, car souvent jugée non rentable, expérimentalement. Par contre, la détection de *clauses unitaires* est un des éléments fondamentaux de tout solveur SAT (y compris pour les solveurs SAT « modernes », dont c'est l'un des éléments clés. Une clause unitaire est une clause de

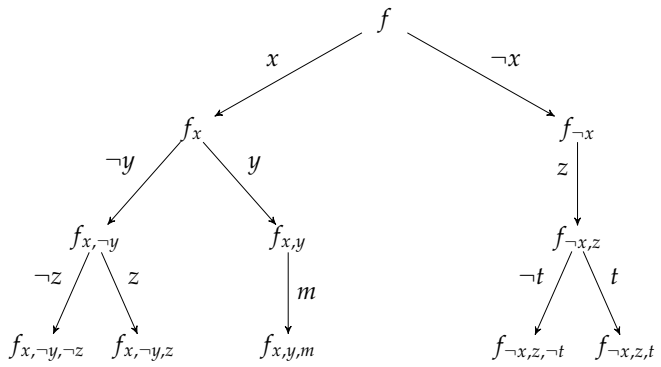


FIGURE 2.1 – *Exemple d'arbre illustrant le parcours arborescent d'une recherche de modèle par un algorithme de type DPLL.*

longueur 1. Cependant, il faut bien noter que cette détection se fait par rapport à l'interprétation partielle courante. Si le solveur a par exemple choisi successivement pour v les littéraux x puis $\neg y$ puis z, une clause initiale $\neg x \vee t \vee y \vee \neg z$ deviendrait soudainement unitaire. Sur une machine récente, et sur un problème industriel typique, on peut mesurer plus d'un million de ces détections par seconde).

Même si l'algorithme est récursif, il faut voir sa trace comme l'exploration d'un arbre dont les feuilles sont des contradictions ou une affectation totale des variables sans contradiction. La figure 2.1 représente la recherche effectuée typiquement par un algorithme de type DPLL. On voit par exemple que le littéral m apparaît dans une clause unitaire lorsque f est simplifiée par l'interprétation partielle $x = \mathtt{V}, y = \mathtt{V}$.

L'autre élément primordial est la **fonction heuristique de choix**, sur laquelle une grande partie de l'efficacité de ces solveurs repose. De nombreuses heuristiques ont été proposées pour tenter de limiter la taille de l'arbre de recherche. L'heuristique la plus connue, appelée MOMS (pour Maximum number of Occurences in Minimum Size clauses) fut introduite dans [55, 84]. Cette heuristique privilégie la variable ayant le plus grand nombre d'occurrences dans les clauses les plus courtes. Dans bien des cas, cependant, elle risque de ne simplifier qu'une seule des deux branches de la recherche. Pour équilibrer les deux sous-arbres, l'heuristique de Jeroslow & Wang [101] estime la variable x à l'aide de la formule $\alpha \times m(x) \times m(\neg x) + m(x) + m(\neg x) + 1$ où $m(x)$ est une mesure de représentativité de x et α une constante (l'heuristique BOHM [39], souvent

utilisée, est un raffinement de MOMS avec cette même idée d'équilibrage des sous-arbres).

Il existe des heuristiques demandant encore plus de calcul. Ainsi, [119] propose de privilégier la variable permettant un grand nombre de propagations unitaires, en cascades (recherchant ainsi le nombre maximum d'*avalanches* [13]), après son affectation, dans les deux sous-arbres. Il s'agit là d'une heuristique à forte anticipation (*lookahead*). Poussée plus loin, [93] a proposé une heuristique de double anticipation qui a donné de bons résultats sur les instances aléatoires.

CDCL : les solveurs « modernes »

En privilégiant le calcul heuristique, les solveurs DPLL, décrits ci-dessus, doivent constamment (après chaque affectation et chaque libération de littéral) mettre à jour de nombreux compteurs : à chaque nœud de l'arbre, on doit en effet être capable d'estimer la présence de tous les littéraux dans la formule simplifiée par l'interprétation partielle courante (des clauses ont été supprimées et des clauses ont été réduites). En cherchant à alléger ces mécanismes, il a été proposé une structure de données permettant une détection paresseuse des clauses unitaires, toujours par rapport à l'affectation courante des variables. Cette structure, appelée *Watched Literals* dans [135], a révolutionné l'application de SAT aux instances industrielles, en autorisant le traitement de problèmes avec plusieurs centaines de milliers de variables. Cette structure de données doit cependant céder une contrepartie de première importance : il n'existe plus aucun moyen de connaître, durant la recherche, le nombre de clauses satisfaites, ou le nombre de clauses binaires (ou même unitaires) relativement à l'affectation courante. Pour pouvoir proposer une heuristique dans ces conditions, les solveurs ont troqué de la visibilité contre de la mémoire. Fonctionnant à l'aveugle, l'intégralité du solveur est passée d'un solveur par anticipation (DPLL62) à un solveur basé sur l'apprentissage, grâce à une analyse précise de son passé, et des conflits rencontrés. Aujourd'hui ces mécanismes sont particulièrement bien cernés [69], et peuvent se résumer à moins de 1000 lignes de code [98] où toutes les techniques utilisées sont profondément interdépendantes. Heuristiques, redémarrages ultra-rapides, sauts arrières non chronologiques, propagations unitaires, gestion des clauses utiles à la suite de la recherche et bien entendu structures de données sont entièrement dédiés à l'apprentissage, et offrent souvent l'image d'un algorithme de plus en plus éloigné de la recherche arborescente binaire classique.

Comme on le voit en lisant la description de l'algorithme 2, le solveur essaye d'atteindre rapidement un conflit (une clause vide), puis apprend

Algorithme 2 : Formulation itérative de DPLL : une formulation CDCL (Conflict-Driven Clause Learning), pleinement tournée vers l'apprentissage.

$\mathcal{I} = \emptyset$, $profondeur = 0$;
tant que *Vrai* **faire**
  Effectuer la Propagation Unitaire (PU) sur $(\Sigma, \mathcal{I})$
  si *un conflit est apparu* **alors**
    si $profondeur = 0$ **alors** Renvoyer UNSAT ;
    C = la clause conflit déduite
    ℓ = l'unique littéral de C affecté à la profondeur du conflit
    $profondeur = max\{profondeur(x) : x \in C/\{l\}\}$
    $\mathcal{I}$ = $\mathcal{I}$ moins tous les littéraux assignés à une profondeur supérieure à $profondeur$
    $(\Sigma, \mathcal{I}) = (\Sigma \cup \{C\}, \mathcal{I}.l)$
  sinon
    si $\mathcal{I}$ *est total* **alors** Renvoyer SAT;
    Choisir un littéral ℓ apparaissant dans $\Sigma|\mathcal{I}$
    $\mathcal{I} = \mathcal{I}.\ell$
    $profondeur = profondeur + 1$

une clause, appelée clause assertive (ou clause FUIP, décrite plus loin), permettant de calculer par ailleurs le niveau de retour arrière nécessaire.

Dans cet algorithme, la Propagation Unitaire est primordiale. Elle consiste à assigner tous les littéraux apparaissant dans des clauses unitaires à vrai. Cette propagation a souvent lieu en cascade, une décision entraînant parfois plusieurs centaines de propagations unitaires (ainsi, décider que x est vrai sur les clauses $\neg x \vee y$, $\neg x \vee \neg y \vee z$ va propager $y = \texttt{V}, z = \texttt{V}$). A la fin de la propagation unitaire, soit la clause vide a été trouvée (un conflit est apparu) soit on a la certitude que plus aucune clause unitaire demeure dans la formule simplifiée.

L'heuristique privilégiera les variables vues dans les analyses de conflits récentes (en pratique, toutes les variables vues verront leur score incrémenté d'une valeur, qui elle-même croît exponentiellement après chaque conflit, en suivant une suite géométrique, généralement de raison 1.05). L'analyse de conflit et la notion de FUIP (pour *First Unique Implication Point*) est primordiale (elle a été montrée optimale dans la taille des sauts arrières [14] et dans la qualité des clauses apprises [15]).

Sur l'exemple de la figure 2.2, la base de clause initiale contient les clauses suivantes, devenues unitaires par rapport à l'affectation courante

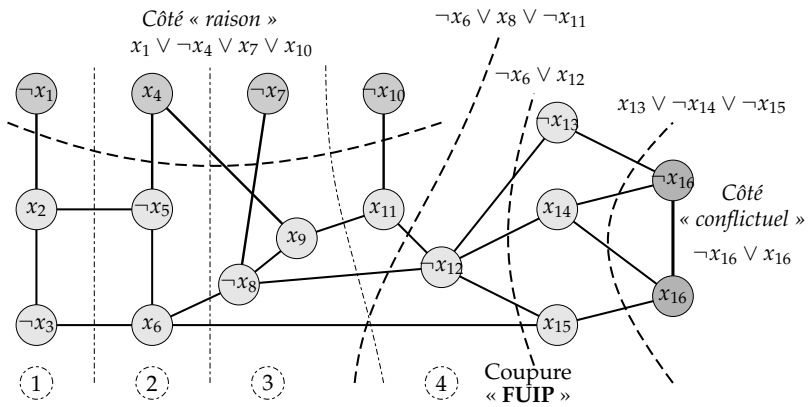


FIGURE 2.2 – *Exemple de graphe d'implication associé aux différents schémas d'apprentissage possibles.*

(entre parenthèses apparaît le niveau de décision auquel elles sont détectées comme étant unitaires) : $x_1 \vee x_2$ et $\neg x_2 \vee \neg x_3$ (niveau 1), $\neg x_4 \vee \neg x_2 \vee \neg x_5$ et $x_5 \vee x_3 \vee x_6$ (niveau 2), $x_7 \vee \neg x_6 \vee \neg x_8$ et $x_8 \vee \neg x_4 \vee x_9$ (niveau 3), ainsi que $x_{10} \vee \neg x_9 \vee x_{11}$, $\neg x_1 \vee x_8 \vee \neg x_{12}$, $x_{12} \vee \neg x_{13}$, $x_{12} \vee x_{14}$, $\neg x_6 \vee x_{12} \vee x_{15}$, $x_{13} \vee \neg x_{14} \vee \neg x_{16}$, et $\neg x_{14} \vee x_{15} \vee x_{16}$ pour le niveau 4. Sur l'exemple donné, on suppose que les différentes variables de division (ou décision) sont, dans l'ordre, les littéraux $\neg x_1$, x_4, $\neg x_7$ et enfin $\neg x_{10}$. Le conflit apparaît lors de la propagation unitaire du quatrième niveau de décision. On représente sur la figure les différentes coupes permettant d'expliquer le conflit, séparant le côté « raison » (les décisions) du côté « conflictuel » (la contradiction). Plusieurs coupures sont possibles, mais une seule contient un littéral FUIP, c'est la *clause assertive* : elle ne contient qu'un seul littéral affecté au dernier niveau de décision (il existe toujours un FUIP, dans la mesure où l'on peut remonter depuis le conflit jusque la dernière variable de décision). On remarquera que dans la clause assertive $\neg x_6 \vee x_{12}$, le troisième niveau de décision n'apparaît plus. Il suffit donc de faire un retour arrière directement au niveau 2, où l'on pourra propager le littéral x_{12}, grâce à la clause assertive nouvellement apprise, devenue unitaire. Le graphe du conflit est un graphe dirigé sans cycle. L'analyse du conflit revient à remonter, en largeur d'abord, depuis le conflit jusqu'à ce que la coupure du graphe calculée ne contienne plus qu'un littéral du dernier niveau de décision. La mise à jour des valeurs heuristiques est effectuée lors de cette phase d'analyse,

en augmentant d'une valeur b toutes les variables vues pendant l'analyse (toutes les variables *à droite* de la coupure). En pratique, la valeur de b est multipliée par une constante, pour se focaliser sur les variables vues dans les derniers conflits (b est en général multiplié par 1.05 à chaque conflit).

De manière théorique, il est intéressant de noter que le parcours en largeur du graphe des conflits revient en fait à effectuer des résolutions entre les clauses responsables de la propagation unitaire des variables du graphe, depuis la clause conflictuelle jusqu'à l'obtention de la clause assertive. Cela a permis à [151] de montrer que les CDCL ont la puissance des systèmes de preuve basés sur la résolution générale. Si, de plus, on ajoute que les solveurs DPLL sont eux basés sur la résolution régulière, on a là des outils théoriques montrant que la puissance des CDCL dépasse celle des DPLL. En effet, il existe dès lors des problèmes difficiles pour les DPLL et faciles pour les CDCL (l'inverse n'est pas vrai).

On ne peut terminer cette courte introduction aux solveurs modernes sans évoquer quelques-uns de leurs composants essentiels, comme les restarts rapides [98, 122] (qui ne sont pas au sens propre des redémarrages, mais plutôt un réordonnancement des dépendances entre variables, puisque les valeurs heuristiques sont conservées : le solveur demeure dans le même espace de recherche), la sauvegarde de phase [150], qui permet de privilégier la dernière polarité des variables lors du branchement, ainsi que la gestion agressive de la base de clauses apprises [15]. Enfin, il faut bien noter, pour conclure sur les solveurs modernes, que beaucoup de choses restent à comprendre dans leurs mécanismes, et que l'on peut encore s'attendre à de nouveaux progrès importants à ce sujet dans les prochaines années. De par leur vélocité et malgré l'apparente simplicité des fonctions heuristiques, les solveurs modernes peuvent s'apparenter à des systèmes complexes, dont le comportement est particulièrement difficile à prévoir et que la communauté cherche, paradoxalement, à mieux comprendre. Ils offrent ainsi d'incroyables performances sur les instances du monde réel, mais comment formaliser les propriétés de telles instances ?

Prétraitement des formules

De manière à combler, en partie, la lacune des solveurs modernes, incapables par exemple de détecter une variable pure, ou des équivalences de littéraux, on leur adjoint quasi systématiquement des techniques de prétraitement. Lors de la compétition SAT 2005, le seul système de prétraitement SATELITE [68] avait ainsi pu résoudre un grand nombre des problèmes industriels proposés. Ces prétraitements sont généralement des applications successives de règles d'oubli de variables (algorithme de DP

limité pour garantir une diminution de la taille de la formule), de résolution hyper-binaire [18], ainsi que de raisonnement sur les clauses bloquées [100]. Même s'ils présentent une *garantie* de rapidité dans le temps maximal passé dans la simplification, leur intégration dans les solveurs CDCL pose problème, dans la mesure où le moindre calcul quadratique dans le nombre de variables serait voué irrémédiablement à l'échec, du fait de la taille parfois gigantesque de certains problèmes. L'élimination des symétries a aussi fait l'objet de nombreux travaux (voir [170], chapitre 6), mais ce n'est que très récemment que les performances dans la détection des symétries ont permis d'espérer embarquer ces procédures dans les solveurs modernes, nécessitant de pouvoir manipuler des instances avec des millions de clauses [104].

Limitations et challenges pour SAT

SAT est un problème qui demande un usage intensif de la mémoire, sujet aux limites des « *memory bound functions* ». Ces fonctions ne permettent pas de suivre les progrès de la loi de Moore sur la rapidité des processeurs, et limitent grandement les possibilités de parallélisation efficace, notamment lors de l'utilisation de plusieurs cœurs. Intuitivement, ces fonctions ont un temps de calcul dominé par le temps passé à lire et/ou écrire en mémoire. Ces algorithmes parcourent de grandes portions de mémoire de manière imprévisible, empêchant tout mécanisme de cache d'être réellement efficace, et limitant la décomposition du problème en plusieurs endroits distincts de la mémoire. Dans ce cadre, et malgré d'importants progrès (voir [170, 91]), la difficulté reste importante et la parallélisation efficace des CDCL représente certainement l'un des plus importants défis pour la communauté SAT toute entière même si de nombreux travaux cherchent à améliorer significativement les performances des SAT solveurs lorsque des milliers de CPUs sont à disposition. Beaucoup reste encore à faire.

2.3 Satisfiabilité Modulo Théories (SMT)

Dans cette partie, nous présentons une extension du problème SAT où les variables booléennes sont remplacées par des contraintes élémentaires (égalités, différences, etc.) entre des termes d'une (union de) théorie(s) du premier ordre (comme l'arithmétique, la théorie de l'égalité, la théorie des tableaux, etc.). Ce problème étendu est appelé SMT (Satisfiabilité Modulo Théories). Pour traiter de telles formules logiques, nous allons devoir modifier l'algorithme CDCL présenté dans la partie précédente afin de le faire coopérer avec des petits moteurs de preuve qui savent prendre en charge ces contraintes.

Dans la suite, nous supposons que le lecteur connaît les notions élémentaires de la logique du premier ordre : syntaxe (signature, termes, formules), modèles (domaines, interprétations), validité logique, théories. Nous rappelons brièvement ces notions dans la section suivante.

2.3.1 Logique du premier ordre : définitions et notations

Nous rappelons ici les définitions et notions élémentaires de la logique du premier ordre.

Signatures et termes. Une *signature* Σ est un ensemble fini de symboles de fonctions et de prédicats. Chaque symbole a une *arité* qui représente le nombre d'arguments auxquels il doit être appliqué. Les *constantes* sont des symboles de fonction d'arité 0. Toute signature Σ est supposée contenir le symbole $=$ de prédicat d'égalité. Traditionnellement, on utilise des lettres en minuscules f, g, h, etc. pour désigner des symboles de fonction, et des lettres en majuscules $P, Q, R, \ldots$ pour les symboles de prédicats.

Soit $\mathcal{V}$ un ensemble de *variables*, distinctes des symboles de Σ. On note $T(\Sigma, \mathcal{V})$ l'ensemble des *termes* associés à Σ, *i.e.* le plus petit ensemble contenant $\mathcal{V}$ et tel que $f(t_1, \ldots, t_n) \in T(\Sigma, \mathcal{V})$ si $t_1, \ldots, t_n \in T(\Sigma, \mathcal{V})$ et $f \in \Sigma$. L'ensemble $T(\Sigma, \emptyset)$ est celui des *termes sans variable* (en anglais, *ground terms*).

Formules. Une *formule atomique* est de la forme $P(t_1, \ldots, t_n)$, où $t_1, \ldots, t_n$ sont des termes de $T(\Sigma, \mathcal{V})$ et P est un symbole de prédicat de Σ. Les *littéraux* sont des formules atomiques (ou leur négation). Les *formules* sont construites inductivement à partir des formules atomiques et de connecteurs booléens ($\wedge$, $\vee$, $\neg$, $\Rightarrow$, etc.) ainsi que des quantificateurs $\forall$ et $\exists$. Une *formule sans variable* (en anglais *ground formula*) ne contient que des *termes*

sans variable. Une variable est *libre* dans une formule si elle n'est liée par aucun quantificateur. Une *formule close* (en anglais *sentence*) est une formule sans variable libre.

Modèles. Un *modèle* $\mathcal{M}$ pour une signature Σ et un ensemble de variables $\mathcal{V}$ est défini par : (1) un domaine $\mathcal{D}_\mathcal{M}$, (2) une interprétation $f^\mathcal{M}$ pour chaque symbole de fonction $f \in \Sigma$, (3) un sous-ensemble $P^\mathcal{M}$ de $\mathcal{D}_\mathcal{M}^n$ pour chaque prédicat $P \in \Sigma$ d'arité n, et enfin (4) un dictionnaire, traditionnellement nommé $\mathcal{M}$ comme le modèle, qui associe à chaque variable $x \in \mathcal{V}$ une valeur $\mathcal{M}(x)$ du domaine $\mathcal{D}_\mathcal{M}$. La *cardinalité* d'un modèle $\mathcal{M}$ est la cardinalité de $\mathcal{D}_\mathcal{M}$.

Sémantique. Étant donnés une signature Σ, un ensemble de variables $\mathcal{V}$ et un modèle $\mathcal{M}$ pour Σ et $\mathcal{V}$, on définit la sémantique des termes par les deux équations suivantes :

$$\begin{array}{lcl} \mathcal{M}[x] & = & \mathcal{M}(x) \\ \mathcal{M}[f(t_1, \ldots, t_n)] & = & f^\mathcal{M}(\mathcal{M}[t_1], \ldots, \mathcal{M}[t_n]) \end{array}$$

Étant donnée une formule ϕ dont les termes sont dans $T(\Sigma, \mathcal{V})$, on définit la valeur de vérité de ϕ par rapport à $\mathcal{M}$ à l'aide d'une relation binaire $\models$ définie de la manière suivante :

$$\begin{array}{lcl} \mathcal{M} \models t_1 = t_2 & = & \mathcal{M}[t_1] = \mathcal{M}[t_2] \\ \mathcal{M} \models P(t_1, \ldots, t_n) & = & (\mathcal{M}[t_1], \ldots, \mathcal{M}[t_n]) \in P^\mathcal{M} \\ \mathcal{M} \models \neg\Phi & = & \mathcal{M} \not\models \Phi \\ \mathcal{M} \models \Phi_1 \wedge \Phi_2 & = & \mathcal{M} \models \Phi_1 \text{ et } \mathcal{M} \models \Phi_2 \\ \mathcal{M} \models \Phi_1 \vee \Phi_2 & = & \mathcal{M} \models \Phi_1 \text{ ou } \mathcal{M} \models \Phi_2 \\ \mathcal{M} \models \forall x.\Phi & = & \mathcal{M}\{x \mapsto v\} \models \Phi \text{ pour tout } v \in \mathcal{D}_\mathcal{M} \\ \mathcal{M} \models \exists x.\Phi & = & \mathcal{M}\{x \mapsto v\} \models \Phi \text{ pour un certain } v \in \mathcal{D}_\mathcal{M} \end{array}$$

On dira qu'une formule Φ est *satisfiable* s'il existe un modèle $\mathcal{M}$ tel que $\mathcal{M} \models \Phi$, sinon Φ est *insatisfiable*. Deux formules Φ_1 et Φ_2 sont *equisatisfiables* si Φ_1 est satisfiable quand Φ_2 est satisfiable, et réciproquement. Une formule Φ est *valide* si $\neg\Phi$ est *insatisfiable*.

Théories. Une *théorie du premier ordre* T sur une signature Σ est un ensemble de formules closes. Une théorie est *cohérente* (en anglais *consistent*) si elle a (au moins) un modèle.

2.3.2 De petits moteurs de preuve

La technique SMT repose sur des *petits moteurs* de preuve qu'on appelle *procédures de décision*. Ces briques logicielles sont des (semi-)algorithmes pour décider la satisfiabilité de formules pour des théories élémentaires comme l'égalité entre termes non-interprétés, l'arithmétique linéaire sur les entiers ou les rationnels, l'arithmétique de Presburger, des structures de données (listes, tableaux, vecteurs de bits), etc.

Voici des exemples de formules pour trois théories : (1) la théorie de l'égalité avec un symbole non-interprété f, (2) la théorie de l'arithmétique linéaire (sur les rationnels) et (3) la théorie des tableaux.

$$(1) \quad f(f(f(x))) = x \wedge f(f(f(f(f(x))))) = x \wedge f(x) \neq x$$

$$(2) \quad x + y = 19 \wedge x - y = 7 \wedge x \neq 13$$

$$(3) \quad a[i \leftarrow x] = b \wedge a = b \wedge b[i] = y \wedge b[i \leftarrow x][j] = y \wedge i = j$$

En pratique, les formules intéressantes à prouver sont souvent formées d'un mélange de symboles appartenant à plusieurs théories. Les exemples suivants mélangent (1) la théorie des tableaux avec celle de l'arithmétique linéaire sur les entiers et (2) les mêmes théories plus celle de l'égalité avec un symbole non interprété f.

$$(1) \quad v[i \leftarrow v[j]][i] \neq v[i] \wedge i + j \leqslant 2j \wedge j + 4i \leqslant 5i$$

$$(2) \quad x + 2 = y \wedge f(a[x \leftarrow 3][y - 2]) \neq f(y - x + 1)$$

Comme on le voit dans ces exemples, ces procédures de décision sont utilisées pour décider la satisfiabilité de *conjonctions* de contraintes (ou prédicats) élémentaires (égalités, inéquations, relations d'ordre, etc.). Nous allons voir dans les deux sous-sections suivantes des procédures de décision pour deux théories : la théorie de l'égalité avec symboles non interprétés et la théorie des inéquations. Nous verrons ensuite des algorithmes pour *combiner* ces petits moteurs de preuve afin de décider des conjonctions de contraintes mélangeant plusieurs théories.

Incrémentalité, backtrackabilité, explications et implications. Comme nous le verrons dans la section 2.3.3, les procédures de décision utilisées par un démonstrateur SMT doivent posséder quatre propriétés : (1) l'*incrémentalité*, (2) la *backtrackabilité*, (3) la *production d'explications* et (4) la détection de *contraintes impliquées*. Les deux premières ne sont pas indispensables, mais elles améliorent l'efficacité du démonstrateur.

1. **Incrémentalité**. La procédure de décision doit pouvoir être appelée successivement sur une suite d'ensembles de contraintes $\mathcal{C}_0 \subset \mathcal{C}_1 \subset \ldots \subset \mathcal{C}_k$ de telle sorte que le traitement d'un ensemble $\mathcal{C}_i$ ne nécessite pas de tout refaire depuis le début, mais réutilise le traitement effectué pour $\mathcal{C}_{i-1}$.

2. **Backtrackabilité.** Les structures de données utilisées par la procédure de décision doivent permettre de revenir efficacement à un état antérieur de l'algorithme sans avoir à tout ré-initialiser.

3. **Explications.** Quand l'ensemble $\mathcal{C}$ de contraintes est insatisfiable, la procédure de décision doit produire un sous-ensemble $\mathcal{U}$ de $\mathcal{C}$ incohérent. Cet ensemble $\mathcal{U}$, appelé en anglais *unsat core*, représente la raison pour laquelle $\mathcal{C}$ est insatisfiable. $\mathcal{U}$ doit être le plus petit possible, c'est-à-dire qu'il doit contenir le moins possible de contraintes redondantes ou inutiles. Par exemple, si $\mathcal{C} = \{x < y, y < z, z < 10, x < z, x > 15, a > b\}$, l'ensemble $\mathcal{U}$ devrait seulement être égal à $\{z < 10, x < z, x > 15\}$.

4. **Contraintes impliquées.** Les procédures de décision doivent enfin permettre de détecter de nouveaux faits (ou contraintes) impliqués par un ensemble de contraintes $\mathcal{C}$. En particulier, comme nous le verrons en section 2.3.3, les procédures devront pouvoir détecter des égalités entre variables.

Théorie de l'égalité avec symboles non interprétés

Cette théorie, appelée également *théorie libre de l'égalité*, donne un sens au prédicat d'égalité $=$ en présence de symboles de fonctions d'arité quelconque. Le sens de chaque symbole de fonction n'est pas défini. Dit autrement, il s'agit là de la théorie *syntaxique* de l'égalité.

Les termes de cette théorie appartiennent à deux catégories syntaxiques :
(1) les variables x, y, z, etc.
(2) les applications de fonctions $f(x), g(x, y)$, etc.

Les contraintes élémentaires sont soit des égalités ou des différences entre les termes comme par exemple $x = f(y, z)$, $g(x) \neq h(y)$ ou $x = y$.

La théorie est définie à partir de trois axiomes et d'un schéma d'axiomes.

(Réflexivité) $\forall x.\ x = x$

(Symétrie) $\forall xy.\ x = y \Rightarrow y = x$

(Transitivité) $\forall xyz.\ x = y \wedge y = z \Rightarrow x = z$

(Congruence) Pour tout symbole de fonction f d'arité n

$$\forall x_1, \ldots, x_n, y_1, \ldots, y_n.$$
$$x_1 = y_1 \wedge \cdots \wedge x_n = y_n \Rightarrow f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n)$$

Les trois premiers axiomes expriment que le prédicat d'égalité est une relation d'équivalence (relation réflexive, symétrique et transitive). Le schéma d'axiomes ajoute le fait que c'est une congruence, c'est-à-dire que pour tout symbole de fonction f, l'égalité se propage aux applications de f pour des arguments deux à deux égaux.

Procédure de décision. La procédure de décision pour cette théorie est appelée *fermeture par congruence*.

Étant donnée une conjonction de contraintes élémentaires $\mathcal{C}$ de la forme $\bigwedge_i t_i \bowtie u_i$, où $\bowtie$ est une contrainte d'égalité $=$ ou de différence $\neq$, l'algorithme consiste tout d'abord à séparer l'ensemble $\mathcal{C}$ en deux ensembles $\mathcal{E}$ et $\mathcal{D}$, contenant respectivement les contraintes d'égalités et de différences.

$$\overbrace{\bigwedge_i t_i = u_i}^{\mathcal{E}} \quad \wedge \quad \overbrace{\bigwedge_j t_j \neq u_j}^{\mathcal{D}}$$

Ensuite, un graphe dirigé acyclique (DAG) est construit pour représenter tous les termes (et sous-termes) apparaissant dans l'ensemble $\mathcal{E} \cup \mathcal{D}$. La boucle principale de la fermeture par congruence est donnée par l'algorithme 3. Elle consiste à maintenir une structure *union-find* pour manipuler des classes d'équivalence entre les nœuds du graphe : deux nœuds n et m étant dans la même classe quand les termes t_1 et u_1 qu'ils représentent respectivement sont égaux soit (1) parce que $t_1 = u_1 \in \mathcal{E}$, soit (2) parce que l'égalité $t_1 = u_1$ est une conséquence de $\mathcal{E}$ et des axiomes de la théorie de l'égalité.

On rappelle qu'une structure *union-find* permet de maintenir une partition d'un ensemble fini à l'aide de deux fonctions :

- `find(`t`)` : renvoie le représentant d'un élément `t`
- `union(`t_1`,`t_2`)` : fusionne les classes d'équivalences de t_1 et t_2

Algorithme 3 : Fermeture par congruence.

```
1 for every nodes n, m ∈ G labeled with the same symbol do
2     if find(n) ≠ find(m) and
3         find (n_i) = find (m_i) for every children of n and m then
4             merge the classes of n and m by union (n,m)
```

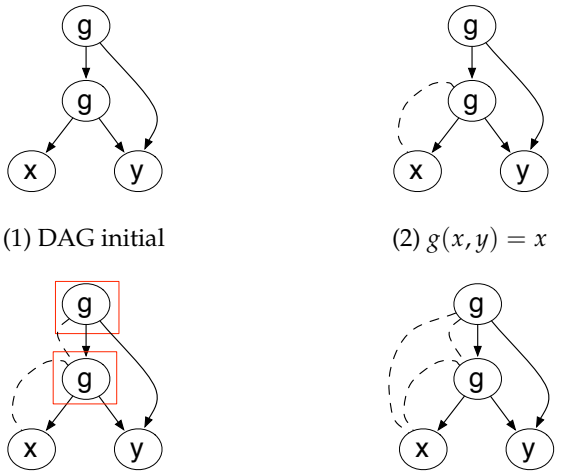
La figure 2.3 illustre le fonctionnement de cet algorithme pour décider la satisfiabilité de la conjonction $g(x, y) = x \wedge g(g(x, y), y) \neq x$. Le DAG construit initialement avec tous les termes et sous-termes du problème est donné en (1).

La structure *union-find* est ensuite initialisée (schéma (2)) avec les nœuds du graphe en mettant dans la même classe les nœuds représentant des termes égaux selon $\mathcal{E}$. Les classes d'équivalences sont matérialisées par des traits en pointillés. Ainsi, le nœud central étiqueté par g (représentant le terme $g(x, y)$) est relié au nœud x car $g(x, y) = x \in \mathcal{E}$.

L'algorithme cherche alors à appliquer l'axiome de congruence. Pour cela, il recherche deux nœuds n et m non encore égaux (donc non reliés par des traits en pointillés) et étiquetés par le même symbole de fonction dont les fils sont égaux deux à deux. Les deux nœuds étiquetés par le symbole `g` (encadrés en rouge dans le schéma (3)) remplissent ces critères et leurs classes d'équivalences sont fusionnées. Cette union de deux classes implique, par transitivité, que les nœuds représentant les termes $g(g(x, y), x)$ et x sont maintenant dans la même classe (schéma (4)).

Une fois la boucle principale terminée, l'algorithme se termine en s'assurant que toutes les différences $t_k \neq t_l \in \mathcal{D}$ sont effectivement vraies dans le graphe, *i.e.* que les nœuds représentant respectivement les termes t_k et t_l ne sont pas reliés par un trait en pointillé. Dans notre exemple, la différence $g(g(x, y), x) \neq x$ n'est pas satisfaite puisque les nœuds représentant ces deux termes sont dans la même classe d'équivalence. On en déduit donc que cette formule n'est pas satisfiable.

Exercice 1 En utilisant la procédure de décision de *fermeture par congruence*, déterminer la satisfiabilité de la formule suivante : $f(f(f(x))) = x \wedge f(f(f(f(f(x))))) = x \wedge f(x) \neq x$.



(1) DAG initial (2) $g(x,y) = x$

(3) $g(x,y) = x \wedge y = y \Rightarrow g(g(x,y),y) = g(x,y)$ (4) $g(g(x,y),y) = g(x,y) \wedge g(x,y) = x \Rightarrow g(g(x,y),y) = x$

FIGURE 2.3 – *Exemple de fermeture par congruence.*

Incrémentalité, backtrackabilité, explications et implications. Une procédure incrémentale avec génération d'explications est présentée dans l'article de Nieuwenhuis et Oliveras *Proof-producing Congruence Closure* [139].

Théorie des inéquations

Cette théorie, appelée en anglais *Difference Logic*, est un fragment de l'arithmétique linéaire (sur les entiers ou les rationnels) où les contraintes élémentaires sont restreintes à la forme $x - y \leqslant c$, avec x et y deux variables et c une constante numérique. Bien que restrictive, d'autres contraintes peuvent être encodées dans ce fragment. Une solution à un ensemble de telles contraintes est représentée par un dictionnaire σ qui à chaque variable x associe un entier ou un rationnel $\sigma(x)$.

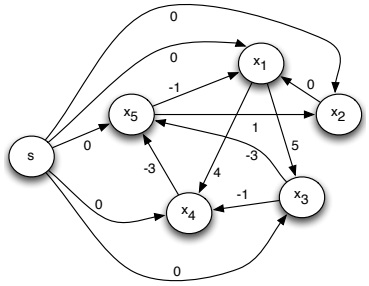
(1) Encodage des contraintes strictes :
- sur $\mathbb{Z}$, $x - y < c$ est remplacée par $x - y \leqslant c - 1$
- sur $\mathbb{Q}$, $x - y < c$ est remplacée par $x - y \leqslant c - \delta$, où δ est une constante suffisamment petite (en pratique, δ est simplement une constante *symbolique*)

(2) Encodage des contraintes de la forme $x \leqslant c$:
Pour cela, on remarque que pour toute solution σ d'un ensemble de contraintes, on peut construire une nouvelle solution σ' en décalant $\sigma(x)$ par une même constante k, pour tout x.
Ainsi, $x \leqslant c$ est encodée par $x - y_{zero} \leqslant c$, où y_{zero} est une nouvelle variable, et pour toute solution σ, on construit une solution σ' telle que $\sigma'(y_{zero}) = 0$, *i.e.* on décale de $k = -\sigma(y_{zero})$.

Procédure de décision. Étant donné un ensemble de variables $\{x_1, \ldots, x_n\}$ et une conjonction $\mathcal{C}$ de contraintes élémentaires de la forme $\bigwedge_i x_i - y_i \leqslant c_i$, la procédure de décision pour cette théorie consiste à construire un graphe pondéré $\mathcal{G}(V, E)$ tel que :
- $V = \{s, x_1, \ldots, x_n\}$, chaque nœud correspond à une variable du problème plus une nouvelle variable s
- $E = \{y_i \xrightarrow{c_i} x_i \mid x_i - y_i \leqslant c_i \in \mathcal{C}\} \cup \{s \xrightarrow{0} x_i \mid 1 \leqslant i \leqslant n\}$, chaque arête correspond à une contrainte du problème plus une arête de poids nul entre s et chaque variable du problème.

L'exemple de la figure 2.4 illustre la construction du graphe $\mathcal{G}(V, E)$ pour l'ensemble $\mathcal{C}$ de contraintes donné à gauche.

$x_1 - x_2 \leqslant 0$
$x_1 - x_5 \leqslant -1$
$x_2 - x_5 \leqslant 1$
$x_3 - x_1 \leqslant 5$
$x_4 - x_1 \leqslant 4$
$x_4 - x_3 \leqslant -1$
$x_5 - x_3 \leqslant -3$
$x_5 - x_4 \leqslant -3$



(1) Ensemble $\mathcal{C}$ d'inéquations (2) Graphe $\mathcal{G}(V, E)$

FIGURE 2.4 – *Initialisation du graphe des inéquations.*

Dans la suite, on appelle *chemin* entre deux nœuds a et b, une séquence d'arêtes de la forme $a \xrightarrow{c_1} v_1 \xrightarrow{c_2} \cdots \xrightarrow{c_{n-1}} v_{n-1} \xrightarrow{c_n} b$ et le *poids* de ce chemin est la somme $c_1 + \cdots + c_n$. Parmi tous les chemins entre a et b, on distingue le *plus court chemin*, *i.e.* le chemin ayant le poids le plus petit.

La procédure de décision pour cette théorie repose sur le théorème suivant.

Théorème 1. *Étant donnée une conjonction d'inéquations $\mathcal{C}$ de la forme*

$$\bigwedge_i x_i - y_i \leqslant c_i$$

et $\mathcal{G}(V, E)$ le graphe correspondant :

1. *Si $\mathcal{G}$ a un cycle* négatif, *i.e. un chemin de la forme $v_1 \xrightarrow{c_1} v_2 \xrightarrow{c_2} \ldots \xrightarrow{c_{n-1}} v_n \xrightarrow{c_n} v_1$ tel que $c_1 + c_2 + \cdots + c_{n-1} + c_n < 0$, alors $\mathcal{C}$ est insatisfiable.*
2. *Sinon, une solution est $x_1 = \delta(s, x_1), \ldots, x_n = \delta(s, x_n)$, où $\delta(s, x_i)$ est le* chemin le plus court *entre s et x_i.*

Preuve. (1) Tout cycle négatif $v_1 \xrightarrow{c_1} v_2 \xrightarrow{c_2} \ldots \xrightarrow{c_{n-1}} v_n \xrightarrow{c_n} v_1$ correspond à un ensemble de contraintes :

$$\begin{aligned} v_2 - v_1 &\leqslant c_1 \\ v_3 - v_2 &\leqslant c_2 \\ &\ldots \\ v_1 - v_n &\leqslant c_n \end{aligned}$$

Si on additionne ces inéquations, on obtient que $0 \leqslant c_1 + c_2 + \cdots + c_n$ ce qui contredit l'hypothèse du cycle négatif $c_1 + c_2 + \cdots + c_n < 0$.
(2) Si $\mathcal{G}(V, E)$ n'a pas de cycle négatif alors pour toute arête $y_i \xrightarrow{c} x_i$ on a $\delta(s, x_i) \leqslant \delta(s, y_i) + c$, ou de manière équivalente $\delta(s, x_i) - \delta(s, y_i) \leqslant c$. Ainsi, en posant $x_i = \delta(s, x_i)$ et $y_i = \delta(s, y_i)$ on satisfait la contrainte $x_i - y_i \leqslant c$.

□

La détection de cycles négatifs peut être réalisée à l'aide d'un algorithme de plus court chemin. Un algorithme bien connu est celui de Bellman-Ford (voir l'algorithme 4) qui calcule les plus courts chemins à partir d'un sommet source dans un graphe orienté pondéré.

Cet algorithme maintient une *borne supérieure* $d[x]$, pour chaque nœud x, qui représente le poids du plus petit chemin de s à x. Le cœur de l'algorithme consiste à mettre à jour $d[x]$ en utilisant une technique de *relaxation* qui, pour chaque arête $y \xrightarrow{c} x$, teste si on peut améliorer le plus court chemin $d[x]$ vers x trouvé jusque-là en passant par y. Les chemins sont eux stockés dans un tableau π tel que $\pi[x]$ contient le prédécesseur de x. La figure 2.5 donne le résultat de cet algorithme sur l'exemple précédent.

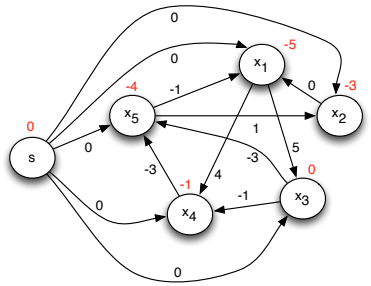
Preuve de correction. Supposons que $\mathcal{G}(V, E)$ contienne un cycle négatif $v_0 \xrightarrow{c_0} v_1 \xrightarrow{c_1} \ldots \xrightarrow{c_{k-1}} v_k$ avec $v_0 = v_k$. Supposons que l'algorithme de

Algorithme 4 : Algorithme de Bellman-Ford.

```
1  for each x_i ∈ V do
2      d[x_i] := ∞;
3  d[s] ← 0;
4  for i = 1 to |V| − 1 do
5      for each y_i --c--> x_i ∈ E do
6          if d[x_i] > d[y_i] + c then
7              d[x_i] ← d[y_i] + c;
8              π[x_i] ← y_i
9  for each y_i --c--> x_i ∈ E do
10     if d[x_i] > d[y_i] + c then
11         return Negative Cycle Detected (use π to reconstruct the cycle)
```

Bellman-Ford ne trouve pas ce cycle. Ainsi, $d[v_i] \leqslant d[v_{i-1}] + c_{i-1}$ pour tout $i = 1, 2, \ldots, k$. Par sommation de ces inéquations, on obtient $\sum_{i=1}^{k} d[v_i] \leqslant \sum_{i=1}^{k} d[v_{i-1}] + \sum_{i=1}^{k} c_{i-1}$, c'est-à-dire $\sum_{i=1}^{k} d[v_i] - \sum_{i=1}^{k} d[v_{i-1}] \leqslant \sum_{i=1}^{k} c_{i-1}$. Mais, puisque $v_0 = v_k$, on a $\sum_{i=1}^{k} d[v_i] = \sum_{i=1}^{k} d[v_{i-1}]$. Donc, $0 \leqslant \sum_{i=1}^{k} c_{i-1}$, ce qui est impossible puisque ce cycle est supposé être négatif.

□



$$\begin{aligned} x_1 &= -5 \\ x_2 &= -3 \\ x_3 &= 0 \\ x_4 &= -1 \\ x_5 &= -4 \end{aligned}$$

FIGURE 2.5 – *Plus courts chemins et solution de l'ensemble de contraintes de la figure 2.4.*

Exercice 2 En utilisant l'algorithme de Bellman-Ford, déterminer la satisfiabilité de $x \leqslant 1, x - y \leqslant 2, y - z \leqslant 3, z - x \leqslant -6$.

Incrémentalité, backtrackabilité, explications et implications. Ces propriétés pour cette procédure de décision sont discutées et étudiées dans l'article de Nieuwenhuis et Oliveras *DPLL(T) with Exhaustive Theory Propagation and Its Application to Difference Logic* [138].

2.3.3 Interfacer SAT et procédures de décision

Dans cette section, on s'intéresse au problème SMT en se restreignant à des formules d'une *unique* théorie T disposant d'une procédure de décision Tsolve. Nous verrons dans la section suivante comment traiter le problème plus général avec plusieurs théories.

Combinaison hors ligne

Étant donnée une formule ϕ, la technique la plus simple pour résoudre le problème SMT est de convertir ϕ en une forme normale disjonctive (DNF) $\bigvee_{i \in I} \mathcal{C}_i$ puis d'appeler $\texttt{Tsolve}(\mathcal{C}_i)$ sur chaque conjonction de contraintes $\mathcal{C}_i$, jusqu'à ce qu'une de ces contraintes soit T-satisfiable.

Plutôt que de faire cette conversion en DNF, nous allons utiliser un solveur SAT pour traiter efficacement la structure booléenne de ϕ. L'algorithme 5 implémente un solveur SMT dit *hors ligne* (*offline* en anglais).

Algorithme 5 : Algorithme SMT hors ligne.

```
1  function SmtOffline(φ) begin
2      f ← T2B(φ);
3      while True do
4          (res, M) ← CDCL(f);
5          if res = UNSAT then
6              return UNSAT
7          else
8              (res, uc) ← Tsolve(B2T(M));
9              if res = SAT then
10                 return SAT
11             else
12                 f ← f ∧ ¬T2B(uc)
```

L'algorithme commence par calculer une abstraction booléenne de ϕ (supposée être en forme normale conjonctive – CNF) à l'aide de la fonction T2B. Cette transformation consiste simplement à remplacer chaque littéral

de ϕ par une variable booléenne fraîche (une implémentation efficace de T2B cherchera à réaliser un partage maximal de ces variables). La formule f ainsi obtenue est alors passée à un solveur SAT CDCL. L'appel à $\texttt{CDCL}(f)$ renvoie un couple (res, M) où res indique si f est satisfiable et, si c'est le cas, M contient un modèle booléen pour cette formule. Ce modèle est ensuite passé à la procédure de décision Tsolve afin de s'assurer qu'il est cohérent modulo la théorie T. Pour cela, il faut d'abord retrouver les littéraux de départ à l'aide d'une fonction B2T. L'appel à $\texttt{Tsolve}(\texttt{B2T}(M))$ renvoie un couple (res, uc) où res indique si le modèle est satisfiable modulo T. Si $res = SAT$ alors la formule ϕ est donc satisfiable modulo T. Dans le cas contraire, il faut « bloquer » ce modèle booléen pour que le SAT solveur cherche un autre modèle. Pour cela, on utilise l'explication (*unsat core*) uc renvoyée par la procédure de décision afin de recommencer la recherche de modèle sur la formule $f \wedge \neg\texttt{T2B}(uc)$.

L'exemple de la figure 2.6 illustre le fonctionnement du solveur SMT hors ligne sur la formule suivante :

$$\begin{gathered} \neg(a = b) \\ (x = a \vee x = b) \\ (y = a \vee y = b) \\ (z = a \vee z = b) \\ \neg(x = y) \end{gathered}$$

Le principal avantage d'un solveur SMT hors ligne est qu'il permet de réutiliser un solveur SAT « sur l'étagère », sans rien à avoir à modifier. Le solveur SAT communique avec le solveur SMT uniquement à travers les variables booléennes manipulées par les fonctions T2B et B2T. Cette technique permet donc de profiter des avancées les plus récentes sur les solveurs SAT. Malheureusement, cette modularité peut aussi s'avérer être un désavantage. En effet, le solveur SAT n'étant pas guidé par la théorie pour rechercher un modèle, il fait des choix de variables ou des déductions logiques (BCP [1]) « à l'aveugle », pouvant l'amener à explorer des parties de l'espace de recherche inutiles où il se perd complètement.

CDCL(T)

Pour remédier aux problèmes des solveurs SMT hors ligne, on va intégrer directement le raisonnement modulo théorie dans l'algorithme CDCL [1]. Cette procédure SMT, appelée CDCL(T), est donnée dans l'algorithme 6.

1. Voir partie sur SAT.

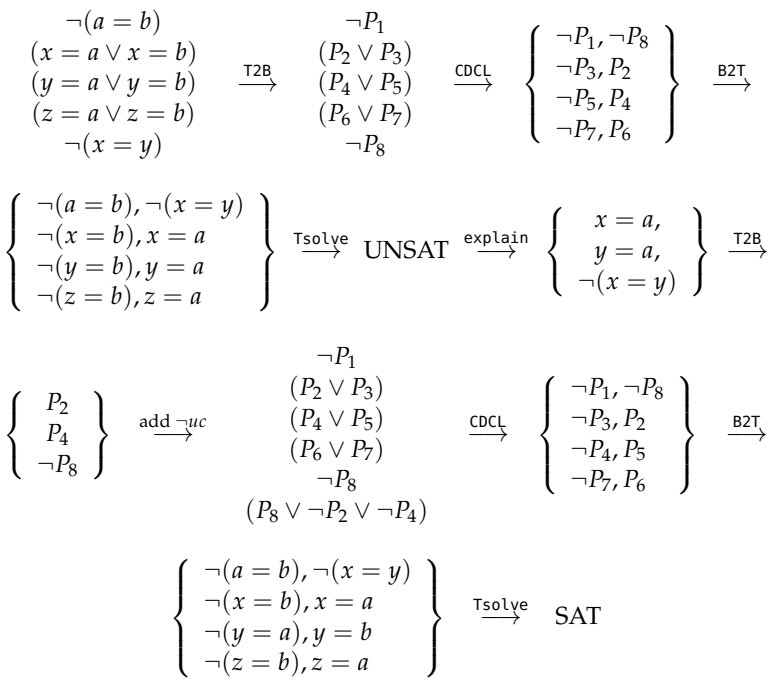


FIGURE 2.6 – *Fonctionnement du SMT hors ligne.*

L'extension de l'algorithme CDCL au raisonnement modulo théorie nécessite peu de modifications. Tout d'abord, sans que cela soit explicite dans l'algorithme, on suppose que toutes les structures de données du solveur SAT sont adaptées pour manipuler directement des littéraux avec des symboles de fonctions et de prédicats.

Le premier mécanisme à étendre est celui de la déduction booléenne (BCP). Celle-ci est réalisée par une fonction `theory_and_boolean_propagation`(ϕ,μ) (ligne 5). Cette fonction réalise d'abord la phase de déduction BCP, puis, si aucun conflit n'est détecté, elle appelle la procédure de décision de la théorie `T` pour vérifier la cohérence du modèle μ modulo `T`. Cette phase de déduction peut également être complétée par la propagation de littéraux déduits par la procédure de décision de la théorie. Ces littéraux pourront à leur tour permettre de déduire de nouvelles clauses unitaires pour continuer la boucle de déduction.

Le deuxième mécanisme à modifier est celui de l'analyse de conflits. La fonction `theory_and_boolean_conflict_analysis` (ligne 13) utilise les

Algorithme 6 : Algorithme CDCL(T).

```
1  function cdclT(φ) begin
2      μ ← [];
3      dl ← 0;
4      while True do
5          res ← theory_and_boolean_propagation(φ,μ);
6          if res = SAT then
7              if all_variables_are_assigned(φ,μ) then
8                  return SAT
9              l ← pick_a_branching_literal(φ,μ);
10             dl ← dl + 1;
11             push(μ,l@dl)
12         else
13             (lvl, cls) =
                theory_and_boolean_conflict_analysis(φ,μ);
14             if lvl < 0 then
15                 return UNSAT
16             backtrack(φ, μ, lvl);
17             learn(cls);
18             dl ← lvl
```

explications renvoyées par la procédure de décision pour trouver le niveau et la clause conflit modulo la théorie.

Comme la procédure de décision est maintenant partie prenante des choix et rebroussements du solveur SAT, il est fondamental qu'elle possède les propriétés d'incrémentalité et de backtrackabilité décrites dans la section 2.3.2.

L'exemple décrit dans la figure 2.7 illustre le fonctionnement de l'algorithme CDCL(T) sur un problème SMT. Le tableau à double colonnes décrit les différentes étapes de l'algorithme. Les quatre premières lignes indiquent des choix de littéraux de branchement (correspondant aux lignes 9 - 11 dans CDCL(T)). La ligne suivante correspond à une propagation modulo théorie où le littéral $(x_1 - x_2) > 3$ est détecté par la procédure de décision. Cette déduction permet de déclencher de nouvelles propagations booléennes (BCP). Cela amène à trouver un conflit entre les trois littéraux $(x_1 - x_3 > 6)$, $(x_3 = x_5 + 4)$ et $(x_1 - x_5 \leqslant 1)$, ce qui conduit à calculer la clause conflit $x_1 - x_3 \leqslant 6 \vee (x_3 \neq x_5 + 4) \vee (x_2 - x_3 > 2)$. Cet enchaîne-

$$(x_2 - x_3 \leqslant 2) \vee A_1$$
$$\neg A_2 \vee (x_1 - x_5 \leqslant 1)$$
$$(x_1 - x_2 \leqslant 3) \vee A_2$$
$$(x_3 + x_4 < 5) \vee (x_1 - x_3 > 6) \vee \neg A_1$$
$$(x_1 - x_2 \leqslant 3) \vee A_1$$
$$(x_2 - x_4 \leqslant 6) \vee (x_5 = 5 - 3x_4) \vee \neg A_1$$
$$(x_3 = x_5 + 4) \vee A_1 \vee A_2$$

@1	$(x_1 - x_3 > 6)$
@2	$(x_3 = x_5 + 4)$
@3	$(x_2 - x_4 \leqslant 6)$
@4	$(x_2 - x_3 \leqslant 2)$
T-propagate	$(x_1 - x_3 > 6) \wedge (x_2 - x_3 \leqslant 2) \Rightarrow (x_1 - x_2 > 3)$
BCP	A_1
BCP	A_2
BCP	$(x_1 - x_5 \leqslant 1)$
T-conflict	$(x_1 - x_3 > 6) \wedge (x_3 = x_5 + 4) \wedge (x_1 - x_5 \leqslant 1)$
backtrack	$x_1 - x_3 \leqslant 6 \vee (x_3 \neq x_5 + 4) \vee (x_2 - x_3 > 2)$
keep@1	$(x_1 - x_3 > 6)$
keep@2	$(x_3 = x_5 + 4)$
BCP	$(x_2 - x_3 > 2)$
...	

FIGURE 2.7 – *Fonctionnement de CDCL(T).*

ment illustre l'intérêt de combiner les raisonnements booléen et théorie au sein de l'algorithme CDCL. L'algorithme poursuit alors en rebroussant au niveau de décision @2 et en ajoutant par BCP le littéral $x_2 - x_3 > 2$ (voir dans la partie SAT le mécanisme de calcul d'une clause conflit en appliquant le principe de résolution).

2.3.4 Combinaison de procédures de décision

Dans la section précédente, nous avons décrit l'algorithme CDCL(T) qui combine un solveur SAT avec une *unique* procédure de décision pour une théorie T. En pratique, les formules à vérifier impliquent souvent plusieurs théories. Dans cette section, on s'intéresse aux techniques pour combiner plusieurs procédures de décision.

L'algorithme de Nelson-Oppen

Un des principaux problèmes sous-jacents à l'implémentation d'un solveur SMT est le problème de la construction d'une procédure de décision pour l'union de théories. Ce problème peut être défini de la manière suivante :

Si $\mathcal{T}_1$ et $\mathcal{T}_2$ sont deux théories *cohérentes* [2] :

1. Est-ce que l'union $\mathcal{T}_1 \cup \mathcal{T}_2$ est cohérente ?
2. Peut-on construire une procédure de décision $\mathcal{T}_1 \cup \mathcal{T}_2$ à partir de procédures de décision pour $\mathcal{T}_1$ et $\mathcal{T}_2$?

Bien qu'il n'y ait pas de réponse dans le cas général, des résultats ont été donnés pour certaines classes de théories. Par exemple, quand $\mathcal{T}_1$ et $\mathcal{T}_2$ n'ont pas de symboles communs (*i.e.* quand leurs signatures sont disjointes), le théorème suivant de Tinelli et Harandi [191] (corollaire 3.3 page 9) apporte une réponse à cette question.

Théorème 2. *Soient deux théories cohérentes et disjointes $\mathcal{T}_1$ et $\mathcal{T}_2$, si $\mathcal{T}_1$ et $\mathcal{T}_2$ admettent toutes les deux un modèle de cardinalité infinie [2], alors l'union $\mathcal{T}_1 \cup \mathcal{T}_2$ est cohérente.*

Preuve. Par le théorème ascendant de Löwenhein-Skolem [94, 174] (qui dit pour résumer que si une théorie possède un modèle infini, elle possède un modèle de n'importe quelle cardinalité infinie), on peut choisir pour $\mathcal{T}_1$ et $\mathcal{T}_2$, respectivement, deux modèles $\mathcal{A}_1$ et $\mathcal{A}_2$ de même cardinalité infinie. Supposons par l'absurde que l'union $\mathcal{T}_1 \cup \mathcal{T}_2$ ne soit pas cohérente. Par le théorème de *cohérence jointe*, il doit exister une formule Φ dans l'intersection de ces deux théories telle que $\mathcal{T}_1 \models \Phi$ et $\mathcal{T}_2 \models \neg\Phi$. Puisque $\mathcal{T}_1$ et $\mathcal{T}_2$ sont disjointes, Φ ne peut contenir que des égalités $x = y$ ou différences $x \neq y$ entre variables. Un résultat bien connu en théorie des modèles est que deux modèles de la théorie vide (théorie sans symbole de fonction ou de prédicat autre que $=$) de même cardinalité sont *isomorphes*. Par conséquent, les réduits des modèles $\mathcal{A}_1$ et $\mathcal{A}_2$ à la théorie vide sont isomorphes et ils sont donc soit tous les deux des modèles de Φ, ou aucun des deux ne l'est. Ceci contredit notre hypothèse. □

Une fois que l'on sait que l'union de théories est cohérente pour une certaine classe de théories, il reste à trouver un moyen de combiner les

2. Voir définition en section 2.3.1

procédures de décision pour ces théories. Cependant, cela n'est pas possible en général puisqu'il existe des théories indécidables qui sont l'union de théories décidables (voir [28]).

Même dans le cas simple de théories *disjointes* ci-dessus, la combinaison n'est pas une question triviale. Voyons cela sur un exemple.

Prenons deux théories $\mathcal{T}_1$ et $\mathcal{T}_2$, où $\mathcal{T}_1$ est la théorie de l'arithmétique linéaire et $\mathcal{T}_2$ la théorie de l'égalité avec symboles non interprétés. Soit Φ la formule suivante :

$$(\Phi) \quad f(x) - x = 0 \wedge f(2x - f(x)) \neq x.$$

Une première idée pour décider la satisfiabilité de cette formule consiste à appliquer l'algorithme suivant :

1. décomposer Φ en deux formules logiques *pures* [3] Φ_1 et Φ_2,
2. conclure que Φ est satisfiable si et seulement si Φ_1 et Φ_2 sont prouvées satisfiables en utilisant les procédures de décision de $\mathcal{T}_1$ et $\mathcal{T}_2$, respectivement.

La première étape de l'algorithme, dite de « purification », consiste à appliquer récursivement la méthode suivante : si a est un sous-terme de Φ qui contient uniquement des symboles d'une théorie, alors remplacer a par une variable fraîche z et ajouter l'équation $z = a$ à Φ_i. Après ce mécanisme d'abstraction par variables, Φ est composée de deux formules *pures* Φ_1 et Φ_2 :

$$\begin{array}{ll} (\Phi_1) & z_1 = f(x) \wedge z_2 = f(x) \wedge f(z_3) \neq x \\ (\Phi_2) & z_1 - x = 0 \wedge 2x - z_2 = z_3. \end{array}$$

Il est facile de voir que Φ_1 et Φ_2 sont satisfiables dans $\mathcal{T}_1$ et $\mathcal{T}_2$, respectivement. Cependant, la conjonction $\Phi_1 \wedge \Phi_2$ ne l'est pas. En effet, de $z_1 = f(x)$ et $z_2 = f(x)$, on déduit que $z_1 = z_2$. Maintenant, $z_1 - x = 0$ implique $z_1 = x$, et à partir de $2x - z_2 = z_3$ on déduit la classe d'équivalence $z_1 = z_2 = z_3 = x$. Au final, par congruence, on déduit que $f(x) \neq z_2$ ce qui contredit $f(x) = z_2$. Par conséquent, cette méthode naïve de combinaison ne définit pas une procédure de décision.

Cet exemple illustre une propriété importante de la combinaison de procédures de décision : la non-satisfiabilité d'une telle conjonction $\Phi_1 \wedge \Phi_2$ est localisée dans une formule close, appelée *interpolant*, qui ne contient que des symboles communs aux deux théories. Dans l'exemple précédent, les signatures des deux théories étant disjointes, cet interpolant ne peut

3. Une formule est *pure* si elle ne contient que des symboles d'une seule signature

être constitué que d'égalités (ou différences) entre variables. Un interpolant pour la conjonction $\Phi_1 \wedge \Phi_2$ ci-dessus est par exemple $x = z_1 \wedge x = z_3$.

Ce résultat découle du théorème de Robinson et Craig [174, 94].

Théorème 3 (Cohérence jointe)**.** *Étant données deux théories* cohérentes $\mathcal{T}_1$ *et* $\mathcal{T}_2$*, l'union* $\mathcal{T}_1 \cup \mathcal{T}_2$ *est incohérente si et seulement s'il existe une formule close* Φ *ne contenant que des symboles communs à* $\mathcal{T}_1$ *et* $\mathcal{T}_2$ *telle que* $\mathcal{T}_1 \models \Phi$ *et* $\mathcal{T}_2 \models \neg\Phi$*.*

La plupart des algorithmes pour combiner des procédures de décision implémentent des techniques *efficaces* pour calculer de tels interpolants. C'est le cas en particulier de l'algorithme inventé par G. Nelson et D. Oppen [137].

Cet algorithme s'applique à des théories (de signatures) *disjointes*, c'est-à-dire ne partageant aucun symbole (de fonction ou de prédicat) autre que celui d'égalité. Les théories doivent également avoir la propriété d'être *stables à l'infini* (*stably infinite* en anglais), c'est-à-dire que toute formule satisfiable dans ces théories doit l'être au moins pour un modèle de cardinalité infinie. De nombreuses théories ont cette propriété. Par exemple, l'arithmétique linéaire ou non-linéaire, la théorie de l'égalité avec symboles non interprétés ou la théorie des tableaux. Par contre, certaines théories comme la théorie des vecteurs de bits ou celle des types énumérés ne l'ont pas.

Le cadre théorique sur lequel s'applique la méthode de Nelson-Oppen étant fixé, l'algorithme 7 décrit son fonctionnement pour des théories *convexes*. Une théorie est convexe si pour toute formule ϕ, lorsque ϕ implique une disjonction d'égalités entre variables, alors ϕ implique nécessairement une de ces égalités. Plus formellement,

$$\begin{array}{c} \text{Si } \mathcal{T} \models \forall \vec{x}.(\phi \Rightarrow x_1 = y_1 \vee \ldots \vee x_n = y_n) \text{ alors} \\ \text{il existe } 1 \leqslant i \leqslant n \text{ tel que } \mathcal{T} \models \forall \vec{x}.(\phi \Rightarrow x_i = y_i). \end{array}$$

Les théories de l'arithmétique linéaire sur les réels et l'égalité avec symboles non interprétés ont cette propriété. Par contre, l'arithmétique linéaire sur les entiers ne la possède pas. Par exemple, dans cette théorie, pour toute variable x, il est vrai que $1 \leqslant x \leqslant 2 \Rightarrow x = 1 \vee x = 2$, mais aucune des deux égalités n'est impliquée.

La propriété de convexité pour une théorie `T` permet de définir une interface pour sa procédure de décision `Tsolve` : pour toute conjonction de contraintes ϕ, `Tsolve`(ϕ) renvoie un couple (res, eq) où *res* indique si ϕ est satisfiable modulo `T` et, si c'est le cas, *eq* contient l'ensemble des égalités

Algorithme 7 : Algorithme de Nelson-Oppen pour deux théories convexes.

```
1  function NO(φ) begin
2      (φ1, φ2) ← purification(φ);
3      while True do
4          (res1, eq1) ← Tsolve1(φ1);
5          (res2, eq2) ← Tsolve2(φ2);
6          if res1 = UNSAT or res2 = UNSAT then
7              return UNSAT
8          if eq1 = eq2 = ∅ then
9              return SAT
10         φ1 ← φ1 ∧ eq1 ∧ eq2;
11         φ2 ← φ2 ∧ eq1 ∧ eq2;
```

$x = y$ impliquées par ϕ, entre des variables contenues dans ϕ, et telles que $x = y \notin \phi$.

L'algorithme de Nelson-Oppen commence par purifier la formule ϕ en entrée. Il entre ensuite dans une boucle pour faire coopérer les deux procédures de décision par échange d'égalités. Les deux formules ϕ_1 et ϕ_2 obtenues par purification sont envoyées aux procédures de décision $\texttt{Tsolve}_1$ et $\texttt{Tsolve}_2$, respectivement. Si l'une des deux formules est insatisfiable, alors $NO(\phi)$ renvoie UNSAT. Dans le cas contraire, si les procédures de décision ne renvoient aucune nouvelle égalité impliquée, l'algorithme s'arrête en concluant que ϕ est satisfiable. Autrement, les deux formules ϕ_1 et ϕ_2 sont renforcées avec la conjonction $eq_1 \wedge eq_2$ et la boucle recommence.

L'exemple suivant illustre le fonctionnement de l'algorithme de Nelson-Oppen pour deux théories convexes (l'arithmétique linéaire et la théorie de l'égalité). La formule Φ sur laquelle est appliqué l'algorithme est la suivante :

$$f(x) - x = 0 \wedge f(2x - f(x)) \neq x.$$

Le tableau de la figure 2.8 donne les étapes de la boucle principale de l'algorithme après que la phase de purification de Φ ait produit les deux formules ϕ_1 et ϕ_2 suivantes :

$$\begin{aligned} \phi_1 &\equiv z_1 = f(x) \wedge z_2 = f(x) \wedge f(z_3) \neq x \\ \phi_2 &\equiv z_1 - x = 0 \wedge 2x - z_2 = z_3. \end{aligned}$$

(1)	ϕ_1	$z_1 = f(x) \wedge z_2 = f(x) \wedge f(z_3) \neq x$
	ϕ_2	$z_1 - x = 0 \wedge 2x - z_2 = z_3$
	(res_1, eq_1)	SAT, $z_1 = z_2$
	(res_2, eq_2)	SAT, $z_1 = x$
(2)	ϕ_1	$z_1 = f(x) \wedge z_2 = f(x) \wedge f(z_3) \neq x \wedge \mathbf{z_1 = z_2} \wedge \mathbf{z_1 = x}$
	ϕ_2	$z_1 - x = 0 \wedge 2x - z_2 = z_3 \wedge \mathbf{z_1 = z_2} \wedge \mathbf{z_1 = x}$
	(res_1, eq_1)	SAT, $\emptyset$
	(res_2, eq_2)	SAT, $x = z_3$
(3)	ϕ_1	$z_1 = f(x) \wedge z_2 = f(x) \wedge f(z_3) \neq x \wedge z_1 = z_2 \wedge z_1 = x \wedge \mathbf{x = z_3}$
	ϕ_2	$z_1 - x = 0 \wedge 2x - z_2 = z_3 \wedge z_1 = z_2 \wedge z_1 = x \wedge \mathbf{x = z_3}$
	(res_1, eq_1)	UNSAT, –
	(res_2, eq_2)	SAT, –

FIGURE 2.8 – *Fonctionnement de l'algorithme de Nelson-Oppen.*

Le tour de boucle (1) donne les valeurs de ϕ_1 et ϕ_2 en entrée de boucle, ainsi que les couples (res_1, eq_1) et (res_2, eq_2) renvoyés par les procédures de décision. Les deux formules étant satisfiables dans leurs théories respectives, elles sont renforcées par la conjonction de nouvelles égalités impliquées $z_1 = z_2 \wedge z_1 = x$. Dans le tour de boucle (2), la nouvelle formule ϕ_1 est toujours satisfiable, mais aucune nouvelle égalité n'est impliquée. La formule ϕ_2 est elle aussi satisfiable, mais une nouvelle égalité $x = z_3$ est trouvée par la procédure de décision de l'arithmétique. En ajoutant cette égalité à ϕ_1, le troisième tour de boucle termine l'algorithme après avoir détecté que ϕ_1 est maintenant insatisfiable.

Correction de l'algorithme. La terminaison de l'algorithme et sa sûreté (si UNSAT est renvoyé, c'est bien que ϕ est UNSAT) sont immédiates puisqu'il ne peut y avoir qu'un nombre fini de nouvelles égalités entre variables et que chaque égalité ajoutée est une conséquence logique de ϕ_1 ou ϕ_2.

La preuve de complétude de l'algorithme consiste à montrer que si $\texttt{Tsolve}_1(\phi_1)$ et $\texttt{Tsolve}_2(\phi_2)$ renvoient tous les deux SAT, alors $\phi_1 \wedge \phi_2$

est bien satisfiable pour l'union des deux théories. Soient $\mathcal{M}_1$ et $\mathcal{M}_2$, les modèles de ϕ_1 et ϕ_2, respectivement. Pour que ϕ soit satisfiable dans l'union des théories, il suffit de combiner $\mathcal{M}_1$ et $\mathcal{M}_2$ en un modèle $\mathcal{M}$ qui soit non seulement un modèle de l'union des théories, mais aussi un modèle de $\phi_1 \wedge \phi_2$. Cette construction s'appelle une *amalgamation*.

Pour montrer que cette amalgamation est possible, il suffit de montrer (voir [94, 174]) qu'il existe une partition $\mathcal{D}$ de l'ensemble des variables partagées entre ϕ_1 et ϕ_2 qui soit compatible avec ϕ_1 et ϕ_2.

La construction de $\mathcal{D}$ s'appuie sur l'invariant de boucle suivant : ϕ_1 et ϕ_2 sont de la forme $\phi_1 \equiv \phi_1' \wedge \psi$ et $\phi_2 \equiv \phi_2' \wedge \psi$, où ψ est une conjonction d'égalités entre variables et ϕ_1' (resp. ϕ_2') ne contient aucune égalité entre variables (la preuve est laissée en exercice). Ainsi, si l'algorithme renvoie SAT, alors puisque $eq_1 = eq_2 = \emptyset$, on en déduit que *pour tout couple* de variables (x, y), ϕ_1 implique (modulo $\mathcal{T}_1$) $x = y$ si et seulement si ϕ_2 implique (modulo $\mathcal{T}_2$) $x = y$. On définit alors $\mathcal{D}$ de sorte que deux variables x et y partagées sont dans la même classe d'équivalence *si et seulement si* $\models \psi \Rightarrow x = y$. Supposons maintenant que ϕ_1 (resp. ϕ_2) ne soit pas compatible avec $\mathcal{D}$. Cela veut dire qu'il existe deux variables partagées x et y telles que $\mathcal{D}(x) \neq \mathcal{D}(y)$ et $\phi_1 \Rightarrow x = y$. Par construction de $\mathcal{D}$ et l'invariant de boucle de l'algorithme, ce cas est impossible. □

L'algorithme de Nelson-Oppen a deux points faibles. Le premier est qu'il nécessite d'instrumenter les procédures de décision pour déduire les égalités impliquées par une formule. Le deuxième est que le traitement des théories *non-convexes* nécessite d'une part d'étendre le mécanisme de déduction pour générer des disjonctions d'égalités, et d'autre part, de traiter ces disjonctions au sein même de l'algorithme de la combinaison de théories, alors que ces disjonctions seraient mieux traitées par un solveur SAT. Les deux sections suivantes présentent des variantes de cet algorithme pour éviter ces deux écueils.

L'analyse par cas à la demande

Cette technique nécessite d'étendre l'interface de la théorie avec une fonction `splitting_on_demand` pour ajouter à la formule d'entrée des clauses d'analyse par cas [20]. L'algorithme 8 montre la modification à apporter à `cdclT()` : il suffit d'appeler cette nouvelle fonction après avoir propagé les déductions booléennes et modulo théories. Le solveur SAT pourra ainsi être utilisé pour réaliser des analyses par cas.

Il est important de remarquer que les clauses d'analyse par cas peuvent contenir des termes ou littéraux non présents dans la formule de départ.

L'ajout de nouveaux éléments peut évidemment compromettre la terminaison de l'algorithme mais également sa sûreté. Pour garantir que la fonction `cdclT` s'arrête, on impose que l'ensemble des nouveaux littéraux des clauses d'analyse par cas soit fini. Pour garantir sa sûreté, il faut contraindre la procédure de décision à ne produire que des clauses qui préservent l'équisatisfiabilité à chaque tour de boucle. Ces restrictions ne semblent pas être une limitation pour les théories utilisées en pratique.

Algorithme 8 : CDCL(T) avec lemmes d'analyse par cas.

```
1  function cdclT(φ) begin
2      μ ← [];
3      dl ← 0;
4      while True do
5          res ← theory_and_boolean_propagation(φ,μ);
6          if res = SAT then
7              if all_variables_are_assigned(φ,μ) then
8                  return SAT
9              splitting_on_demand(φ,μ);
10             ...
11         else
12             ...
```

La combinaison retardée (DTC)

Cette approche, appelée en anglais *Delayed Theory Combination* [37, 33], consiste à faire coopérer directement les procédures de décision avec le solveur SAT, ce dernier se chargeant de l'échange des égalités (entre variables d'interface) entre les procédures.

L'algorithme 9 peut être vu comme une intégration de l'algorithme de Nelson-Oppen pour combiner deux théories $\mathcal{T}_1$ et $\mathcal{T}_2$ dans l'algorithme CDCL. Après avoir purifié la formule en entrée ϕ en deux formules ϕ_1 et ϕ_2, l'algorithme DTC initialise une variable α contenant l'ensemble des atomes sur lesquels effectuer des décisions, ainsi que toutes les égalités entre les variables partagées par ϕ_1 et ϕ_2. La variable μ contient la pile des décisions et des littéraux impliqués.

La boucle `repeat` implémente, en séquence, la déduction des contraintes booléennes et modulo les théories $\mathcal{T}_1$ et $\mathcal{T}_2$. La propagation des contraintes booléennes est classique. Pour déduire de nouveaux

Algorithme 9 : Delayed Theory Combination.

```
1  function dtc(φ) begin
2      μ ← [];
3      (φ1, φ2) ← purification(φ);
4      α ← atoms(φ1 ∧ φ2, interface_equalities(φ1, φ2));
5      while True do
6          repeat
7              res ← boolean_propagation(φ1 ∧ φ2,μ);
8              if res = UNSAT then
9                  (lvl, cls) = boolean_conflict_analysis(φ1 ∧ φ2,μ);
10                 if lvl < 0 then return UNSAT;
11                 backjump_and_learn(φ, μ, cls,lvl);
12             else
13                 μ1^e = extract_theory1_and_interface_equalities(μ);
14                 res1 ← Tsolve1(φ1, μ1^e);
15                 if res1 = UNSAT then
16                     (lvl, cls1) = theory1_conflict_analysis(φ1,μ1^e);
17                     if lvl < 0 then return UNSAT;
18                     backjump_and_learn(φ, μ, cls1, lvl);
19                 else
20                     μ2^e = extract_theory2_and_interface_equalities(μ);
21                     res2 ← Tsolve2(φ2, μ2^e);
22                     if res2 = UNSAT then
23                         (lvl, cls2) = theory2_conflict_analysis(φ2, μ2^e);
24                         if lvl < 0 then return UNSAT;
25                         backjump_and_learn(φ, μ, cls2, lvl);
26                     else
27                         if all_variables_are_assigned(α) then return SAT;
28                         theory1_splitting_on_demand(φ1, μ1^e);
29                         theory2_splitting_on_demand(φ2, μ2^e);
30         until fixpoint on μ;
31         ℓ ← pick_a_branching_literal(α);
32         decide_on(μ, ℓ)
```


littéraux impliqués modulo la théorie $\mathcal{T}_1$, on doit d'abord extraire les faits de cette théorie, ainsi que les contraintes d'égalités entre variables partagées, qui ont été ajoutées dans la pile μ. L'ensemble obtenu μ_1^e est passé en argument à la procédure de décision de la théorie $\mathcal{T}_1$ (avec la formule ϕ_1). En cas de conflit, on extrait une clause de conflit modulo $\mathcal{T}_1$ et on rebrousse dans l'arbre de décision. La propagation des contraintes modulo $\mathcal{T}_2$ est identique. Si tous les littéraux déduits sont satisfiables et que tous les atomes contenus dans α ont une valeur de vérité, on arrête l'algorithme en renvoyant SAT. Sinon, on ajoute à la fin de la boucle des lemmes d'analyse par cas, comme vu précédemment. Lorsqu'on atteint le point fixe de cette propagation de contraintes, le solveur SAT ajoute un nouveau littéral de décision dans μ. Ce littéral est choisi parmi l'ensemble α, qui rappelons-le, contient toutes les égalités entre variables partagées. Ainsi, le solveur SAT construit bien une partition entre ces variables, d'une manière similaire à l'algorithme de Nelson-Oppen.

La combinaison dirigée par les modèles

Un des inconvénients de l'algorithme DTC est que le solveur SAT doit gérer un nombre quadratique d'égalités entre variables partagées. Il choisit également en aveugle parmi ces égalités.

Pour remédier à ces deux problèmes, la combinaison dirigée par les modèles (*model based theory combination* [58]) étend l'algorithme DTC afin de permettre aux procédures de décision de déduire *simplement* des égalités. Mais contrairement à l'algorithme de Nelson-Oppen, les égalités déduites par une procédure de décision sont celles impliquées par un modèle *candidat* de cette procédure.

Par exemple, si la conjonction de contraintes suivante est donnée à la théorie de l'arithmétique linéaire sur les entiers :

$$0 \leqslant x \leqslant 1 \wedge 0 \leqslant y \leqslant 1 \wedge z = y - 1$$

la procédure de décision peut construire le modèle $[x \mapsto 0; y \mapsto 0; z \mapsto -1]$ et déduire l'égalité $x = y$.

Bien sûr, ces égalités peuvent s'avérer être en contradiction avec les autres théories et nécessiter de rebrousser. Dans l'exemple précédent, il est tout à fait possible que le modèle pour cet ensemble de contraintes soit celui où x et y prennent des valeurs différentes.

Ainsi, pour permettre la gestion (avec rebroussement) de ces égalités dirigées par les modèles, il suffit de les générer lors de l'appel à la fonction `theory`i`_splitting_on_demand`. Ainsi, une égalité impliquée $x = y$ sera

donnée sous la forme d'une clause binaire $x = y \vee x \neq y$, en donnant une priorité de décision au littéral $x = y$.

2.3.5 Pour aller plus loin

On renvoie le lecteur intéressé par une formalisation rigoureuse et complète de l'algorithme CDCL(T) à l'article de Nieuwenhuis et al. [140]. L'algorithme de Nelson-Oppen, ainsi que les preuves de correction, sont également donnés dans plusieurs articles, voir par exemple [191, 50].

Il existe aussi d'autres schémas de combinaison de procédures de décision implantés dans les solveurs SMT. Parmi eux, l'algorithme de R. Shostak [176, 167] inventé au début des années 1980. Cette méthode permet de combiner la théorie de l'égalité avec symboles non interprétés avec une autre théorie, dite de Shostak, disposant d'une interface réduite à deux fonctions :

— un *canonizer*, pour mettre les termes de la théorie en forme normale ;
— un *solver*, pour résoudre des équations pures de la théorie et renvoyer la solution sous la forme d'une substitution (des variables de l'équation vers des termes).

Ces deux fonctions sont alors utilisées pour étendre l'algorithme de clôture par congruence que nous avons vu en section 2.3.2. La combinaison de plusieurs (canonizers et solvers de) théories de Shostak est discutée dans [173, 112].

Un autre schéma de combinaison, appelé MCSAT, a été introduit récemment par L. de Moura et D. Jovanovic [59]. Cette méthode de combinaison met au même plan le solveur SAT et toutes les procédures de décision. L'idée principale est que l'interface de chaque solveur doit implanter une phase de construction de modèles (affectation de valeurs aux variables de son domaine, propagation de contraintes) et une phase de résolution (pour la résolution des conflits modulo théorie). Ainsi, ces procédures coopèrent en intervenant *directement* dans une structure de données partagée appelée *trail*.

On renvoie également le lecteur aux articles [89, 67, 102] pour une présentation approfondie du traitement de l'arithmétique linéaire.

Enfin, nous recommandons la lecture de trois livres qui couvrent une très grande partie de ce domaine de recherche :

— *Handbook of Satisfiability* [25]
— *The Calculus of Computation* [34]
— *Decision Procedures* [111]

2.4 Conclusion

Les solveurs SAT et SMT sont devenus incontournables en informatique, au point qu'on peut réellement parler de la *révolution SAT et SMT*. Ces outils sont aujourd'hui utilisés dans tant de domaines qu'une liste exhaustive serait très longue. Pour n'en donner que quelques-uns, citons l'intelligence artificielle, la conception de micro-processeurs, la vérification de logiciels, la synthèse de programmes, etc.

Cette révolution est avant tout liée à l'augmentation incroyable des performances de ces démonstrateurs qui sont maintenant capables de traiter des problèmes de taille industrielle.

Dans ce cours, nous avons présenté les principales techniques et algorithmes des démonstrateurs SAT et SMT modernes. Le lecteur qui souhaite approfondir ce sujet pourra trouver des compléments d'information dans les références bibliographiques contenues dans ces notes de cours.