



HAL
open science

Fragments de l'arithmétique dans une combinaison de procédures de décision

Diego Caminha Barbosa de Oliveira

► **To cite this version:**

Diego Caminha Barbosa de Oliveira. Fragments de l'arithmétique dans une combinaison de procédures de décision. Génie logiciel [cs.SE]. Université Nancy II, 2011. Français. NNT : . tel-00578254v2

HAL Id: tel-00578254

<https://theses.hal.science/tel-00578254v2>

Submitted on 27 May 2011

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Fragments de l'arithmétique dans une combinaison de procédures de décision

THÈSE

présentée et soutenue publiquement le 14 mars 2011
pour l'obtention du

Doctorat de l'Université Nancy 2
(spécialité informatique)

par

Diego CAMINHA BARBOSA DE OLIVEIRA

Composition du jury

Rapporteurs:

Pascal GRIBOMONT

Professeur à l'Université de Liège

David MONNIAUX

Chargé de Recherche au CNRS, Verimag

Examineurs:

David DÉHARBE

Professeur à l'Universidade Federal do Rio Grande do Norte

Pascal FONTAINE

Maître de conférences à l'Université Nancy 2 - Codirecteur de thèse

Claude MARCHÉ

Directeur de recherche à l'INRIA Saclay

Stephan MERZ

Directeur de recherche à l'INRIA Nancy - Directeur de thèse

Jeanine SOUQUIÈRES

Professeur à l'Université Nancy 2

Contents

I	Thèse	7
1	Thèse	9
1.1	Introduction	9
1.2	Les solveurs SAT	12
1.2.1	L’algorithme DPLL	13
1.3	Les solveurs SMT	14
1.4	Décider une combinaison de théories	15
1.4.1	La méthode de combinaison de Nelson-Oppen	16
1.4.2	Combinaison par partage d’égalité de modèles	19
1.5	Procédure de décision pour SMT	21
1.5.1	Génération d’ensemble de conflit	21
1.5.2	Génération d’égalité	22
1.5.3	Génération d’égalité de modèle	22
1.5.4	Lemmes	22
1.5.5	Propagation de théorie	22
1.5.6	Incrementalité and backtrackabilité	22
1.6	La logique de différence	23
1.6.1	Procédure de décision pour la logique de difference	25
1.7	Décider l’arithmétique linéaire	27
1.7.1	La méthode du simplexe	27
1.7.2	Décider la satisfaisabilité incrémentalement	30
1.7.3	Génération de l’ensemble de conflit	33
1.7.4	Backtracking	33
1.7.5	Génération d’égalité	34
1.7.6	Génération d’égalités de modèle	34
1.7.7	Diségalités et inégalités strictes	35
1.7.8	Variables entières	35
1.8	Conclusion	36
II	Extended thesis	37
2	Introduction	39

2.1	The overview of the thesis	41
2.2	The publications during the thesis	42
3	From SAT to SMT-solvers	45
3.1	Introduction	45
3.2	SAT-solver	46
3.2.1	The DPLL algorithm	47
3.2.2	DPLL, an example	48
3.2.3	Modern techniques for DPLL based SAT-solvers . . .	49
3.3	An example of modeling in SAT	51
3.4	SMT-solver	53
3.5	An example of modeling in SMT	55
3.6	Conclusion	56
4	Deciding a combination of theories	59
4.1	The Nelson and Oppen combination framework	60
4.1.1	Equality generation and propagation	60
4.1.2	Generation of disjunction of equalities and propagation	63
4.1.3	Trying all arrangements	64
4.1.4	Remarks	66
4.2	Combination sharing model-equalities	67
4.3	Combining with model-equalities, an algorithm	74
4.4	Soundness and completeness of SMT-solvers	77
4.4.1	Model-equalities	80
4.5	Conclusion	82
5	Extending a basic decision procedure	83
5.1	Introduction	83
5.2	Conflict set generation	84
5.3	Equality generation	86
5.4	Model-equality generation	87
5.5	Lemmas	88
5.6	Theory propagation	89
5.7	Incrementality and backtrackability	91
5.8	Conclusion	95
6	Deciding difference logic	97
6.1	Difference logic graph theory	97
6.1.1	Properties and Graph Representation	98
6.1.2	Conclusion	102
6.2	Difference logic decision procedure	102
6.2.1	Satisfiability Checking	103
6.2.2	Incremental Satisfiability Checking	104
6.2.3	Conflict Set Construction	108

6.2.4	Equality Generation	109
6.2.5	Model-Equality Generation	115
6.2.6	Generating fewer model-equalities	118
6.2.7	Theory propagation	122
6.3	Conclusion	124
7	Deciding linear arithmetic	125
7.1	Introduction to the simplex method	125
7.2	The primal simplex	127
7.3	Incremental satisfiability check	134
7.4	The unsatisfiable case	146
7.5	Generating the conflict set	147
7.6	Backtracking	149
7.7	Equality generation	150
7.8	Model-equality generation	151
7.9	Disequalities and strict inequalities	151
7.10	Integer variables	152
7.11	Conclusion and future work	153
8	Conclusion	155

Part I
Thèse

Chapter 1

Thèse

1.1 Introduction

Concevoir des logiciels corrects est un défi majeur actuel. Mais construire un logiciel totalement exempt de bogues est, la plupart du temps, une tâche extrêmement difficile. De nombreuses techniques peuvent être appliquées pendant la conception de logiciels, de façon à ce que ceux-ci fonctionnent du mieux possible. De bonnes pratiques d'ingénierie logicielle pourront aider à atteindre ce but, mais souvent, ce n'est pas suffisant. Le test est un des moyens habituels pour vérifier qu'un programme fonctionne comme attendu, mais il peut être difficile (voire impossible) de tester le programme exhaustivement. Les méthodes formelles s'attachent à prouver formellement des modèles de logiciels. Si le modèle est prouvé, on peut être totalement convaincu que le modèle est correct en regard des propriétés prouvées. Cependant, comme toutes les autres techniques, les méthodes formelles ont leur limitation.

La confiance nécessaire en un logiciel dépend du type de celui-ci. Pour de nombreuses applications l'effort supplémentaire à fournir pour assurer que le logiciel fonctionne à la perfection est prohibitif, et ce coût ne se justifie pas. Les techniques formelles nécessitent effectivement beaucoup de travail et une grande expertise. Pour beaucoup de logiciels, il est suffisant de délivrer régulièrement des corrections : un bogue détecté n'aura pas des conséquences financières ou humaines justifiant des méthodes longues et coûteuses.

Quand le logiciel devient critique pour les vies humaines ou si un problème dans le logiciel a des conséquences financières ou écologiques qui le justifie, il devient nécessaire d'utiliser des méthodes de vérification. Des logiciels dans l'avionique, le transport de personnes sur rails, etc. . . peuvent avoir des conséquences en terme de vies humaines. Il est essentiel de s'assurer de leur robustesse. Des logiciels utilisés dans les banques, ou en bourse peuvent être développés avec des méthodes plus coûteuses, dans l'optique que cela permettra de ne pas perdre de l'argent plus tard, quand le logiciel

sera en fonctionnement. Par ailleurs, du logiciel robuste permet d'obtenir et de garder la confiance des utilisateurs. Un dernier type de logiciel dont la robustesse est importante est le logiciel embarqué, qui, pour certains types, ne peuvent être mis à jour une fois le produit distribué.

Le test est une étape incontournable du développement du logiciel. Cela permet de détecter beaucoup d'erreurs simples et habituelles. Afin d'être sûr que le logiciel est correct, on peut naïvement tester toutes les entrées possibles, et vérifier le comportement sur ces entrées. Cela n'est possible que pour des cas très spécifiques où le nombre de configurations d'entrée est petit. Généralement, le nombre de configurations d'entrée croît exponentiellement avec la taille de l'entrée, et cela implique qu'un test exhaustif prendrait un temps non acceptable même pour des entrées relativement petites. Aussi, dans certaines situations, tester l'ensemble des cas est impossible car l'entrée peut être infinie. Ainsi, pour la plupart des logiciels, des tests doivent être appliqués, mais seulement sur une infime fraction des entrées possibles. Cela permettra de détecter des erreurs, mais cela ne permettra pas de déterminer si un logiciel est sans erreurs.

Les méthodes formelles interviennent ici, pour prouver que le logiciel est correct. Il existe de nombreuses techniques formelles pour montrer que l'implémentation d'un algorithme fonctionne. Mais prouver que tout code, quel que soit le langage de programmation, donne toujours le résultat attendu, n'est pas possible, vu qu'il n'y a aucune théorie formelle qui gère tous les aspects des langages de programmation, des compilateurs, des systèmes d'exploitation, des architectures d'ordinateur, etc. Ainsi, les aspects importants du logiciel sont traduits dans un modèle qui lui, peut être vérifié.

La preuve d'un algorithme peut aussi être conduite au moyen d'outils. Il existe de nombreuses classes d'outils disponibles, interactifs ou automatiques. Quelques exemples : les solveurs SAT, les solveurs SMT, les analyseurs statiques, les assistants de preuve, les model-checkers. L'usage d'un outil ou d'un autre dépend du problème et des spécifications. Dans cette thèse, nous nous concentrons sur les solveurs SMT.

Enfin, les méthodes formelles sont en évolution constante. De nouvelles méthodes sont régulièrement proposées, ainsi que de nouveaux langages de modélisation, et des prouveurs plus robustes. Si les méthodes formelles sont à l'heure actuelle assez peu attractives, elles sont utilisées assez largement dans l'industrie. Elles sont aussi en expansion, puisque un des buts principaux est de les rendre accessibles et utilisables pour de plus en plus d'utilisateurs et d'applications.

Les aspects à prouver impliquent généralement de nombreuses théories comme des théories des listes, des tableaux, des fonctions, des nombres, d'ensembles, etc. Créer un outil qui peut raisonner sur une combinaison de celles-ci n'est pas une chose facile. Il existe une technique classique qui permet de créer une procédure de décision pour une combinaison de théories, en combinant des procédures de décision, chaque procédure combinée gérant

une seule théorie : cette technique est connue sous le nom de combinaison de Nelson-Oppen [52]. Un aperçu assez complet des procédures de décision pour de nombreuses théories peut être trouvé dans [40].

Les solveurs SMT sont parmi les outils les plus en vogue pour la vérification. La plupart des solveurs SMT utilisent des combinaison à la Nelson-Oppen [52]. Ces solveurs sont aussi basés sur des solveurs SAT, qui sont des outils puissants pour résoudre des problèmes Booléens. L'utilisation des solveurs SAT, combinés aux procédures de décision, font des solveurs SMT des outils intéressants pour appréhender des formules issues de la vérifications, dans un langage expressif. Nous aborderons dans cette thèse, la façon dont les différents composants d'un solveur SMT fonctionnent ensemble.

Une théorie généralement utilisée est l'arithmétique. Par exemple, on peut conclure que $x + x = 1$ est satisfaisable sur les réels, mais pas sur les entiers. La théorie arithmétique est extrêmement large et on préfère généralement considérer des fragments avec de meilleures propriétés, les problèmes concrets n'utilisant généralement qu'un sous-ensemble assez petit du langage.

Un fragment particulièrement intéressant est la *logique de différence*. Il est basé sur des contraintes de la forme $x - y \leq c$, où x et y sont des variables et c est une constante numérique. Bien que le langage est très restreint, la logique de différence permet d'exprimer de nombreuses classes de problèmes pratiques, comme des systèmes temporisés, des problèmes d'ordonnement, et des chemins dans des circuits digitaux (voir par exemple [54]). La logique de différence peut être modélisé entièrement en théorie des graphes. Cela permet d'utiliser des algorithmes rapides, capable de résoudre de grandes instances de problèmes.

Un autre fragment arithmétique important est l'arithmétique linéaire. Ce fragment est plus expressif que la logique de différence. Dans ce fragment, les additions, soustractions et multiplications sont autorisées, avec la contrainte que la multiplication doit avoir lieu seulement entre une variable et une constante. Un autre avantage est que le nombre de variables par contrainte n'est pas fixé alors qu'en logique de différence, il faut exactement deux variables par contrainte. Un exemple de contrainte en arithmétique linéaire est $x_1 + 2x_2 - 5x_3 \leq 5$. Les algorithmes pour résoudre des problèmes d'arithmétique linéaire sont moins efficaces que ceux pour résoudre les problèmes de logique de différence, mais il est toujours possible de gérer de gros problèmes en arithmétique linéaire.

Nous étudierons la logique de différence et l'arithmétique linéaire. Nous commencerons par proposer une plate-forme de combinaison basée sur Nelson et Oppen, qui rend plus simple et plus efficace la gestion de problèmes contenant une combinaison de plusieurs théories différentes, incluant de l'arithmétique linéaire. Nous montrerons ensuite comment construire des procédures de décision pouvant être intégrées dans une combinaison, pour

la logique de différence, et plus généralement pour l'arithmétique linéaire.

1.2 Les solveurs SAT

Les solveurs SMT sont construits sur la base d'un solveur SAT. Les solveurs SAT résolvent le problème de la satisfaisabilité Booléenne. Ce problème, aussi connu sous le nom de problème SAT, consiste à déterminer si une formule Booléenne (ou propositionnelle) donnée est satisfaisable, c'est à dire s'il est possible de donner à toutes les variables une valeur de vérité de telle sorte que la formule soit évaluée à vrai.

Le problème de la satisfaisabilité est le premier problème prouvé NP-complet, en 1971 par Cook [18]. Depuis, on a trouvé de nombreux autres problèmes NP-complets. Beaucoup de preuves de NP-complétude sont des preuves par réduction au problème SAT.

Malgré la nature essentiellement exponentielle du problème, des outils sont maintenant capables d'étudier des problèmes SAT très grands. Les solveurs actuels peuvent résoudre des instances avec des millions de variables et plusieurs millions de contraintes. Le succès pratique des solveurs SAT remet en question l'intérêt de considérer la complexité en pire cas, pour ce problème. De fait, quel que soit le domaine, les solveurs SAT peuvent souvent résoudre des problèmes très grands.

Grâce à cette capacité des SAT solveurs à gérer de grands problèmes, beaucoup de problèmes dans des domaines variés sont modélisés en SAT, de façon à bénéficier des bonnes performances des SAT solveurs : vérification logicielle et hardware [12, 69], génération automatique de schémas de test [66, 39], ordonnancement [35], planification [38, 62], algèbre [72], etc. Les solveurs SMT sont aussi un exemple d'utilisation fructueuse des solveurs SAT, vu que le noyau des solveurs SMT est généralement constitué d'un solveur SAT.

Traduire un problème dans le langage du solveur SAT, c'est-à-dire en logique propositionnelle, peut conduire à une augmentation substantielle de la taille du problème. Cependant, la taille n'étant pas le facteur impactant le plus important, les solveurs SAT parviennent à résoudre les problèmes souvent plus rapidement que les techniques traditionnelles.

De nombreuses améliorations ont eu lieu récemment dans le développement des solveurs SAT. Une compétition annuelle de solveurs SAT [10, 9, ?, 57] a contribué de façon importante au développement de techniques astucieuses.

Aujourd'hui, les meilleurs solveurs SAT sont basés sur une variante de la procédure introduite en 1960 par Davis et Putnam [24] et améliorée quelques années plus tard par Davis, Logemann et Loveland [23].

1.2.1 L'algorithme DPLL

L'algorithme DPLL exécute une recherche avec backtracking dans l'espace des assignations partielles. Sa caractéristique principale est d'élaguer l'espace de recherche grâce aux clauses falsifiées. L'algorithme 1 donne la version récursive basique de l'algorithme DPLL.

```

input :  $F$  : CNF_Formula
input :  $p$  : Assignment
output: status : { (SAT, Assignment) ; UNSAT }

// Unit propagation
1 while  $F$  has unit clause  $u \wedge$  contains no empty clause do
2   |  $F := F|_u$ ;
3   |  $p := p \cup \{u\}$ ;
4 end

5 if  $F$  contains the empty clause then return UNSAT ;
6 if  $F$  has no clauses left then return (SAT,  $p$ );

// Branching and decision
7  $\ell :=$  a literal not assigned by  $p$  ;
8 status,  $\Gamma :=$  DPLL( $F|_\ell$ ,  $p \cup \{\ell\}$ );
9 if status = SAT then return status,  $\Gamma$  ;
10 return DPLL( $F|_{\neg\ell}$ ,  $p \cup \{\neg\ell\}$ );

```

Algorithme 1: DPLL récursif.

L'algorithme 1 prend en entrée des formules en CNF¹. L'idée est de répétitivement sélectionner un littéral non assigné ℓ dans la formule d'entrée F et de chercher récursivement une assignation pour la formule $F|_\ell$ ² (assignant ℓ à *vrai*) et $F|_{\neg\ell}$ (assignant ℓ à *faux*). L'étape où ℓ est choisi (ligne 7) est appelée l'étape de branchement. L'étape où ℓ est mis à *vrai* (ligne 8) ou *faux* (ligne 10) est appelée *décision*. La décision est associée à un niveau (de décision) égal à la profondeur de récursion de l'Algorithme 1. La fin de la récursion, qui diminue le nombre de variables assignées, est appelé étape de *backtracking*.

Une assignation partielle p est maintenue pendant toute la recherche. Si $F|_p$ contient la clause vide, la clause correspondante de F est dite *violée* par p . Dans l'étape de *propagation unitaire* (lignes 1 à 4), les clauses unitaires sont immédiatement fixées à *vrai* pour des raisons d'efficacité. Les *littéraux purs* (ceux dont la négation n'apparaît pas) peuvent être assignés à *vrai* en étape de preprocessing et, pour certaines implémentations, pendant la phase

¹CNF signifie *Conjunctive Normal Form*, ou forme normale conjonctive, soit une conjonction de disjonction. Toute formule peut être convertie en CNF en temps linéaire, voir par exemple [15].

²La notation $F|_\ell$ désigne la formule simplifiée obtenue par le remplacement de ℓ par *vrai* et $\neg\ell$ par *faux*, et ensuite l'élimination de toutes les clauses avec au moins un littéral vrai, et la suppression de toutes les occurrences de faux littéraux dans les clauses restantes.

de simplification après chaque branchement.

Les variations de cet algorithme constituent la famille la plus utilisée de procédures complètes pour résoudre le problème SAT. Elles sont fréquemment implémentées d'une façon itérative plutôt que récursive, ce qui implique une utilisation plus faible de la mémoire. L'inconvénient est l'étape supplémentaire nécessaire pour la désassignation des variables au backtracking. L'approche naïve qui requiert d'examiner chaque clause est coûteuse, mais les techniques modernes, comme celles des *two watched literals* permettent de réaliser ceci très rapidement.

L'efficacité des solveurs SAT actuels doit beaucoup à plusieurs techniques récentes. Parmi celles-ci on distingue la propagation rapide grâce au *two watched literals*, les mécanismes d'apprentissage, les stratégies de redémarrage, les stratégies d'élimination de clauses, le backtracking non chronologique, et les heuristiques de décisions.

Bien que ces techniques ne soient pas essentielles pour une compréhension du comportement d'un solveur SAT à l'intérieur d'un solveur SMT, elles sont d'une importance primordiale pour l'efficacité.

1.3 Les solveurs SMT

Les premiers outils permettant de traiter des théories encodent directement tout le problème au niveau propositionnel. La plupart des solveurs SMT utilisent maintenant les solveurs SAT en combinaison avec des procédures de décision capables de raisonner sur les atomes écrits dans divers théories.

De nombreux solveurs SMT ont vu le jour ces dernières années. Certains se spécialisent dans un nombre restreint de théories tandis que d'autres tentent de réunir le plus de théories (et de combinaisons) possibles. La compétition annuelle, la SMT-COMP [3, 1] est un moteur du développement de ces outils.

La communauté SMT met à disposition une librairie de formules banc d'essai, la SMT-LIB [59]. Les utilisateurs des SMT peuvent soumettre leurs propres formules, et les développeurs des SMT peuvent utiliser cette librairie comme une aide au développement de leur outil. Ces formules banc d'essai sont écrites dans un langage standardisé, aussi créé et supporté par la communauté : le langage SMT-LIB 1.2 [58] ou SMT-LIB 2.0 [5]. La plupart des solveurs SMT supportent l'un ou l'autre de ces langages.

La Figure 1.1 présente l'architecture très abstraite d'un solveur SMT. Un solveur SAT est au coeur du solveur SMT. Il sert à générer des assignations propositionnelles. Celles-ci doivent être vérifiées pour déterminer s'il existe une inconsistance au niveau théorie. Dans les solveurs SMT basés sur une combinaison à la Nelson-Oppen, l'assignation est envoyée à un module qui se charge de distribuer les contraintes aux procédures de décision respectives.

Ces dernières vérifient la satisfaisabilité, en se partageant des informations quand plusieurs procédures sont en jeu. Si une inconsistance est découverte, des lemmes sont ajoutés au solveur SAT. Ils invalident l'assignation courante, et force le solveur SAT à proposer une nouvelle assignation. Le travail termine quand une assignation satisfaisable au niveau théorie est trouvée, ou quand le solveur SAT a épuisé toute les assignations propositionnelles.

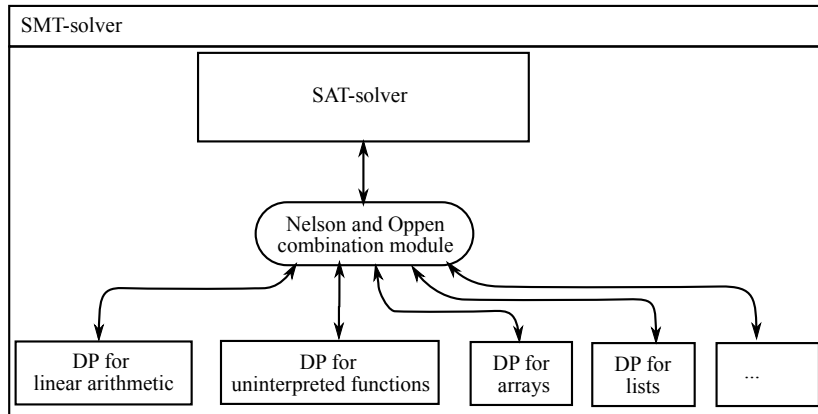


FIG. 1.1: Architecture abstraite d'un solveur SMT basé sur Nelson-Oppen, utilisant plusieurs procédures de décision (DP).

Si une seule théorie est impliquée, il est très simple de réaliser cette intégration entre le solveur SAT et la théorie. Si plusieurs théories sont en jeu, il faut alors recourir à des combinaisons.

1.4 Décider une combinaison de théories

Les formules issues de la vérification de programmes contiennent souvent des termes provenant de différentes théories. Il est plus simple en effet de générer des formules contenant des opérateurs sur plusieurs théories plutôt que d'encoder tout en utilisant une seule théorie. En utilisant plusieurs théories il est d'ailleurs parfois aisé de générer une formule, qui autrement serait difficile, voire impossible à écrire dans un langage plus limité. Quelques exemples de théories habituelles sont l'arithmétique, les symboles non-interprétés, les tableaux, les listes, les ensembles, et les vecteurs de bits.

Il n'est pas simple de vérifier des formules contenant des symboles de plusieurs théories. Les solveurs SMT utilisent des méthodes permettant de combiner des procédures de décisions sur une théorie seulement. Si une procédure conclut à l'insatisfaisabilité, le problème est inconsistant. Cependant, si chacune des théories conclut à la satisfaisabilité, ce n'est pas suffisant pour conclure à la satisfaisabilité de la combinaison. En effet un tout petit

exemple suffit à nous convaincre de ce fait : $x = 0, y = 1 - 1, f(x) \neq f(y)$. La procédure de décision pour l'arithmétique comprend les littéraux $x = 0$ et $y = 1 - 1$, qui sont ensemble, satisfaisables. La procédure de décision pour les fonctions interprétées conclut aussi à la satisfaisabilité de $f(x) \neq f(y)$. Cependant, il est clair que l'ensemble initial est insatisfaisable ; les deux procédures de décisions peuvent déduire cela à condition de s'échanger des informations, comme nous allons le voir dans la suite.

1.4.1 La méthode de combinaison de Nelson-Oppen

La méthode de combinaison de Nelson-Oppen a été présentée pour la première fois dans [52, 53]. Deux décennies plus tard, cette méthode est adoptée dans la plupart des solveurs SMT.

Cette méthode se base sur le fait que si deux théories \mathcal{T}_1 et \mathcal{T}_2 sont disjointes³ et stablement infinies⁴, la satisfaisabilité de $\mathcal{T}_1 \cup \mathcal{T}_2$ peut être déduite de la satisfaisabilité de $\mathcal{T}_1 \cup \mathcal{L}$ et $\mathcal{T}_2 \cup \mathcal{L}$, où \mathcal{L} est un ensemble d'informations partagées. De ce fait, si on a deux procédures de décision, une pour \mathcal{T}_1 , et une autre pour \mathcal{T}_2 , l'effort supplémentaire pour vérifier la satisfaisabilité de $\mathcal{T}_1 \cup \mathcal{T}_2$ est dans la découverte de \mathcal{L} . Ceci peut être étendu à plus de deux théories.

Génération d'égalités et propagation

Le scénario le plus simple correspond à une combinaison de théories convexes uniquement. Une théorie \mathcal{T} est convexe si toute disjonction de littéraux $l_1 \vee l_2 \vee \dots \vee l_n$ telle que $\mathcal{T} \models l_1 \vee l_2 \vee \dots \vee l_n$ est telle que $\mathcal{T} \models l_i$ pour un i entre 1 et n .

De nombreuses théories sont convexes. L'arithmétique linéaire sur les réels, et certaines théories des listes [56] en sont des exemples. Par contre, l'arithmétique linéaire sur les entiers, ou l'arithmétique non linéaire sur les réels, les ensembles, et les tableaux sont non convexes. Nelson et Oppen remarquent que l'ensemble des informations partagées \mathcal{L} peut être déterminé précisément dans le cas de théories convexes, en déduisant et propageant répétitivement les égalités entre variables partagées qui peuvent être déduites de l'ensemble de formules⁵ jusqu'à ce que l'insatisfaisabilité soit déduite, ou qu'aucune égalité ne puisse plus être déduite.

Cela fonctionne comme suit. D'abord des littéraux sont envoyés aux procédures de décision. Chacune reçoit les littéraux relatifs à sa théorie. Si nécessaire, les contraintes doivent être purifiées, en créant de nouvelles variables, de sorte qu'aucune contrainte ne contient de symboles de plus qu'une

³Deux théories sont disjointes si aucun symbole n'apparaît dans les deux théories à la fois, excepté des variables et le symbole d'égalité ($=$).

⁴Une théorie est stablement infinie si toute formule sans quantificateur satisfaisable dans la théorie a un modèle infini.

⁵Les variables partagées sont les variables qui apparaissent dans plus d'une théorie.

théorie. Si l'on conclut à l'insatisfaisabilité, l'ensemble original de littéraux est insatisfaisable. Sinon, les procédures de décision déduisent les égalités conséquences du contexte, et propagent ces égalités aux autres procédures. Avec ces nouvelles contraintes, les procédures peuvent soit décider l'insatisfaisabilité, soit déduire de nouvelles égalités et les propager. La satisfaisabilité peut être déduite quand chacune des procédures conclut à la satisfaisabilité, et qu'aucune égalité ne peut plus être déduite.

Cette méthode est simple, correcte et complète [68]. Il suffit pour l'appliquer que les procédures de décision déduisent les égalités entre variables partagées. Cependant, pour des théories non convexes, les choses se compliquent.

Generation de disjonctions d'égalités et propagation

Considérons la formule 1.1. Elle contient un mélange de théorie non interprétée (\mathcal{T}_{UF}) et de la théorie non convexe⁶ de l'arithmétique linéaire sur les entiers (\mathcal{T}_{LIA}).

$$\phi : x \geq 0 \wedge x \leq 1 \wedge v_1 = 0 \wedge v_2 = 1 \wedge P(x) \wedge \neg P(v_1) \wedge \neg P(v_2) \quad (1.1)$$

La Figure 1.2 donne une simulation de la procédure pour vérifier la satisfaisabilité de la formule 1.1. À l'état 1.0, les contraintes sont données à leur procédure respective. L'insatisfaisabilité n'est pas détectée, et aucune égalité ne peut être déduite. Cependant, la méthode ne peut s'arrêter là, car le problème est évidemment inconsistant. Il faut d'autres techniques pour assurer la complétude de la méthode.

Ce qu'il faut prendre en compte ici, c'est que pour des théories non convexes, bien qu'aucune égalité ne peut être déduite, des disjonctions d'égalités peuvent l'être. Dans cet exemple, la disjonction $x = v_1 \vee x = v_2$ est une conséquence des contraintes, bien que $x = v_1$ et $x = v_2$ ne le soient pas.

$x = v_1 \vee x = v_2$ est une information importante qui manque au solveur pour \mathcal{T}_{UF} . À l'état 1.1, le solveur pour \mathcal{T}_{LIA} génère et propage cette information au solveur pour \mathcal{T}_{UF} . Puisque le solveur pour \mathcal{T}_{UF} est incapable de gérer les disjonctions directement, une étude par cas est nécessaire. Si un chemin sans contradiction peut être trouvé, le problème est satisfaisable. Sinon, si tous les chemins conduisent à une contradiction, le problème est inconsistant. Dans notre exemple, les chemins issus de $x = v_1$ et de $x = v_2$ conduisent tous deux à une contradiction directement après⁷; le problème est donc inconsistant.

⁶Pour se convaincre que l'arithmétique linéaire sur les entiers est non convexe, il suffit de prendre comme exemple $\psi : 0 \leq x \leq 1$. $\psi \implies (x = 0 \vee x = 1)$, mais $\psi \not\Rightarrow x = 0$ et $\psi \not\Rightarrow x = 1$.

⁷ $x = v_1 \wedge P(x) \wedge \neg P(v_1) \implies \perp$; $x = v_2 \wedge P(x) \wedge \neg P(v_2) \implies \perp$.

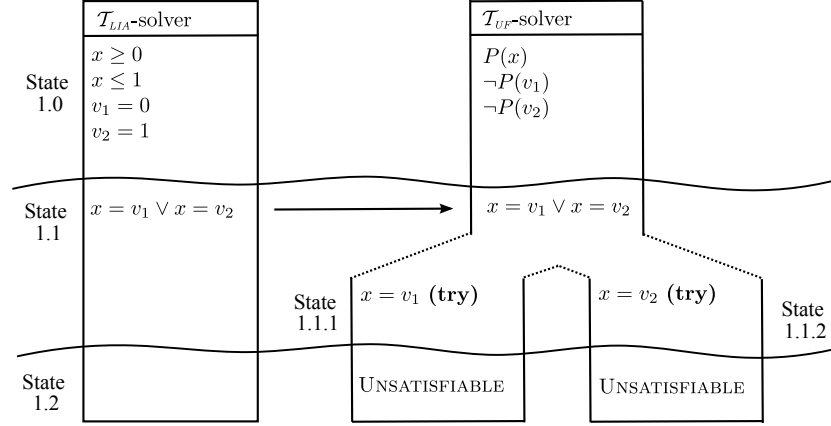


FIG. 1.2: Simulation de Nelson-Oppen sur la Formule 1.1, en propageant des disjonctions d'égalités.

Dans la pratique, les solveurs SMT ne génèrent pas et ne propagent pas des disjonctions d'égalité. C'est une tâche trop complexe. Des techniques particulières sont mises en œuvre.

Tous les arrangements

L'échange de disjonctions d'égalités est une alternative théorique pour les arrangements. Un arrangement, pour Nelson-Oppen, est un ensemble consistant maximal qui, pour chaque paire de variables x, y , contient soit $x = y$, soit $x \neq y$. Une méthode brute, mais complète, de combinaison de théories disjointes et stablement infinies est d'essayer tous les arrangements des variables partagées. Si un de ceux-ci mène à la satisfaisabilité, alors la formule est aussi satisfaisable. Sinon, la formule est inconsistante.

Dans l'exemple précédent (Formule 1.1), il y a trois variables partagées : x , v_1 et v_2 . Il y a cinq arrangements possibles :

$$\begin{aligned}
 \mathcal{A}_1 &= \{x = v_1, x = v_2, v_1 = v_2\} \\
 \mathcal{A}_2 &= \{x = v_1, x \neq v_2, v_1 \neq v_2\} \\
 \mathcal{A}_3 &= \{x \neq v_1, x = v_2, v_1 \neq v_2\} \\
 \mathcal{A}_4 &= \{x \neq v_1, x \neq v_2, v_1 = v_2\} \\
 \mathcal{A}_5 &= \{x \neq v_1, x \neq v_2, v_1 \neq v_2\}
 \end{aligned} \tag{1.2}$$

Précédemment, nous avons vu que pour tester la satisfaisabilité d'une combinaison $\mathcal{T}_1 \cup \mathcal{T}_2$, il est nécessaire de trouver \mathcal{L} tel que le problème peut être réduit à la satisfaisabilité de $\mathcal{T}_1 \cup \mathcal{L}$ et $\mathcal{T}_2 \cup \mathcal{L}$, de façon séparée. Déterminer \mathcal{L} est facile pour des théories convexes, mais infaisable pour des théories non convexes. L'idée derrière les arrangements est que, s'il n'est pas possible

de déterminer \mathcal{L} , essayer tous les arrangements permettra de résoudre le problème. Dans le cas de notre exemple, \mathcal{L} est :

$$\begin{aligned} \mathcal{L} = & (x = v_1 \wedge x = v_2 \wedge v_1 = v_2) \\ & \vee (x = v_1 \wedge x \neq v_2 \wedge v_1 \neq v_2) \\ & \vee (x \neq v_1 \wedge x = v_2 \wedge v_1 \neq v_2) \\ & \vee (x \neq v_1 \wedge x \neq v_2 \wedge v_1 = v_2) \\ & \vee (x \neq v_1 \wedge x \neq v_2 \wedge v_1 \neq v_2) \end{aligned} \quad (1.3)$$

La satisfaisabilité peut être vérifiée en utilisant les procédures de décision et vérifier chaque arrangement sur chaque procédure. Cependant, utiliser un solveur SMT est préférable car on bénéficie alors de l'apprentissage, du backtracking, etc. qui permettent de diminuer grandement la taille de l'espace de recherche. Pour utiliser le solveur SAT sous-jacent au solveur SMT, \mathcal{L} sera convertie en CNF. Le résultat de cette conversion est :

$$\begin{aligned} \mathcal{L}' = & (x = v_1 \vee x \neq v_2 \vee v_1 \neq v_2) \\ & \wedge (x \neq v_1 \vee x = v_2 \vee v_1 \neq v_2) \\ & \wedge (x \neq v_1 \vee x \neq v_2 \vee v_1 = v_2) \end{aligned} \quad (1.4)$$

En reformulant la formule 1.1, on obtient la formule 1.5. Un solveur SMT sans coopération entre les procédures de décision est alors capable de vérifier la satisfaisabilité de la formule.

$$\phi' : \phi \wedge \mathcal{L}' \quad (1.5)$$

1.4.2 Combinaison par partage d'égalité de modèles

Combiner des procédures de décision par propagation d'égalité de modèle [26] est une alternative pour combiner des théories disjointes et stablement infinies, particulièrement appropriée pour des théories non convexes.

Le but de cette combinaison par égalité de modèle est de trouver un arrangement des variables partagée. Plutôt que de deviner un arrangement de façon aveugle, les procédures de décision maintiennent un modèle (ou en construisent un quand nécessaire), et se basent sur ce modèle pour générer des égalités entre variables. Ces égalités générées depuis les modèles sont appelées égalités de modèles. Elles sont propagées aux autres procédures de décision qui les incorporent et continuent le processus habituel.

Considérons l'exemple suivant, dans la théorie arithmétique des entiers :

$$x \geq 0, x \leq 1, v_1 = 0, v_2 = 1 \quad (1.6)$$

Aucune égalité ne peut être déduite. Il existe deux modèles possibles (voir Table 1.1).

Modèle 1		Modèle 2	
Variable	Valeur	Variable	Valeur
x	0	x	1
v_1	0	v_1	0
v_2	1	v_2	1

TAB. 1.1: Modèles pour l'exemple 1.6.

Dans le modèle 1, x et v_1 ont la même valeur. Ainsi, on peut extraire l'égalité de modèle $x = v_1$. Dans le cas du modèle 2, $x = v_2$ peut être extrait.

Une simulation (sur l'exemple 1.1) d'un solveur SMT basé sur la propagation d'égalités de modèle est présentée Figure 1.3. Une différence par rapport à une simulation *examinant successivement tous les arrangements* est qu'il n'y a pas un ensemble complet contenant soit une égalité ou une diségalité pour chaque paire de variables partagées. L'arrangement est créé pendant l'exécution de l'algorithme.

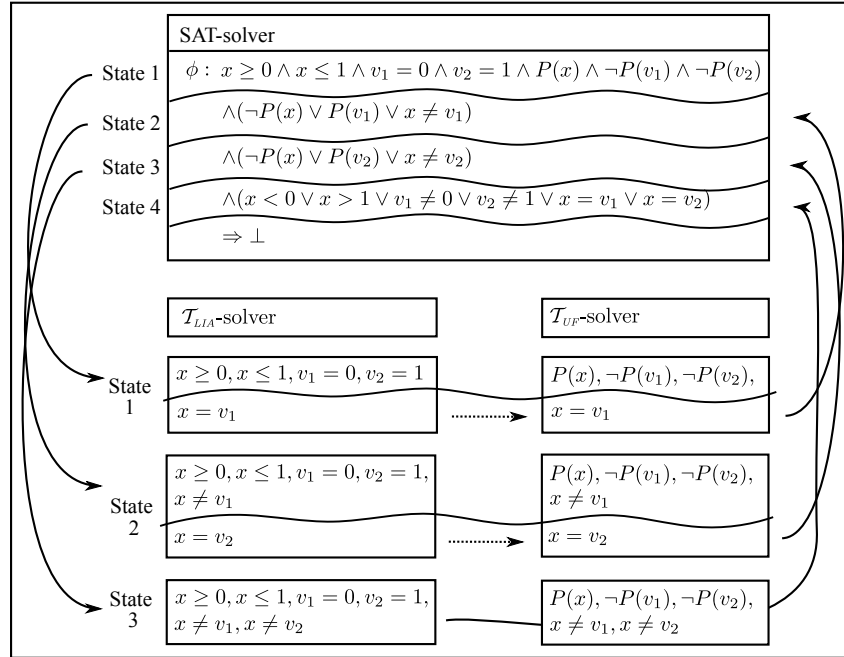


FIG. 1.3: Simulation d'une combinaison par échange d'égalité de modèle (Formule 1.1). Le solveur pour \mathcal{T}_{LIA} génère des égalités de modèle et les propage au solveur pour \mathcal{T}_{UF} . L'inconsistance est déduite à l'état 4, grâce à l'apprentissage.

Un avantage de cette combinaison est que le SAT solveur ne génère pas

de littéraux inconsistants comme $v_1 = v_2$ ⁸. Cela peut contribuer à réduire le nombre d'assignations passées à la procédure de décision pour les théories.

Une autre observation importante est que la non convexité est implicitement traitée par les modèles. Par exemple, la disjonction $x = v_1 \vee x = v_2$ est cachée dans les contraintes arithmétiques. Cependant, pour chaque modèle arithmétique, une seule de ces égalités sera propagée, c'est-à-dire qu'il n'y a aucun modèle tel que $x \neq v_1 \wedge x \neq v_2$ est vrai.

En pratique la combinaison utilisant des égalités de modèle se rapproche de près de la combinaison utilisant uniquement des égalités. Il est très simple d'adapter un solveur SMT pour qu'il gère les égalités de modèle. La différence principale est que si aucune égalité ne peut être déduite, et si l'inconsistance n'est pas détectée, les procédures pour les théories non convexes génèrent des égalités de modèle et les propagent.

Un autre changement en pratique est que ces égalités de modèle peuvent (et le sont souvent) ne pas être présentes dans le problème original. Des ensembles de conflits passés au solveur SAT peuvent, s'ils contiennent des égalités de modèle, contenir de nouveaux littéraux. Cela n'a aucune influence sur la terminaison de la procédure car le nombre de ces nouveaux littéraux est borné par une fonction du nombre de variables dans la formule d'entrée.

1.5 Procédure de décision pour SMT

Une procédure de décision résout un problème où la réponse est binaire : le problème est satisfaisable ou ne l'est pas. Dans notre cas, la procédure de décision décide si une formule arithmétique est satisfaisable ou non, c'est-à-dire s'il existe une interprétation des variables qui rend vraie la formule.

Cependant, dans notre contexte, une procédure doit fournir plus d'information, si elle veut coopérer avec d'autres procédures de décision dans une combinaison, ou si elle doit travailler avec le solveur SAT sous-jacent.

Certaines fonctionnalités sont très importantes pour obtenir un solveur SMT complet pour certaines (combinaisons de) théories. Certaines autres sont importantes pour l'efficacité.

1.5.1 Génération d'ensemble de conflit

Si une procédure détecte l'insatisfaisabilité d'un ensemble donné de contraintes, nous appelons *ensemble de conflit* tout sous-ensemble inconsistant de l'ensemble donné. L'ensemble de conflit peut être l'ensemble donné au complet, mais il est généralement préférable d'obtenir un petit sous-ensemble. Si l'ensemble de conflit est minimal, c'est-à-dire si l'ensemble privé d'une de ses contraintes devient satisfaisable, l'information rendue est optimale. Les ensembles de conflit sont utilisés pour réfuter les assignations

⁸Le solveur \mathcal{T}_{LIA} , sachant que $v_1 = 0$ et $v_2 = 1$, ne générera jamais $v_1 = v_2$.

propositionnelles fournies par le solveur SAT.

1.5.2 Génération d'égalité

La combinaison de Nelson et Oppen requiert des procédures qu'elles soient capables de détecter et propager les égalités entre variables. Il sera aussi nécessaire de construire, une fois l'insatisfaisabilité déduite, un ensemble de conflit qui tient compte des égalités qui ont été générées. Plus précisément, l'ensemble de conflit ne doit pas contenir ces égalités générées mais les littéraux qui permettent de déduire cette égalité.

1.5.3 Génération d'égalité de modèle

En présence de variables entières, la théorie arithmétique linéaire n'est plus convexe. Pour des théories non convexes, propager les égalités déduites n'est pas suffisant pour la complétude de la combinaison. Il sera nécessaire de générer des égalités de modèle.

1.5.4 Lemmes

Les procédures devant transmettre une information au solveur SAT peuvent le faire en utilisant des lemmes. N'importe quelle tautologie peut être ainsi ajoutée à la formule de départ. L'ajout de lemmes pourra augmenter l'efficacité, ou aider la procédure de décision à atteindre la complétude.

1.5.5 Propagation de théorie

Une technique utilisée par les solveurs SMT pour augmenter les performances est la propagation de théories [55]. L'algorithme classique de DPLL déduit uniquement des faits propositionnellement. La propagation de théories utilise aussi les théories pour faire de la propagation, comme au niveau Booléen. Plutôt que de décider aveuglément les littéraux à assigner dans le solveur SAT, le raisonneur de théorie informe le solveur SAT des littéraux qui sont vrais dans l'ensemble des littéraux utilisés dans la formule et non propagés ou décidés au niveau Booléen. De cette façon, l'espace de recherche du solveur SAT peut être grandement réduit.

1.5.6 Incrementalité and backtrackabilité

L'incrémentalité est la capacité de recevoir de l'information à chaque étape et d'être capable de poursuivre un raisonnement sans recommencer de rien. Il est important que les procédures de décision soient incrémentales car il arrive très souvent qu'elles reçoivent de nouvelles contraintes, qui doivent être incorporées avec les anciennes.

L'incrémentalité est souvent considérée de pair avec la backtrackabilité. Il s'agit d'être capable de revenir à une étape précédente, en récupérant l'état, et redémarrer un nouveau raisonnement à partir de cet état. L'incrémentalité et la backtrackabilité doivent être considérés du point de vue de la complexité en temps et en mémoire.

1.6 La logique de différence

L'arithmétique est une théorie extrêmement large. Généralement, les formules impliquant de l'arithmétique n'utilisent qu'un fragment très restreint de la théorie. Il est dès lors utile de considérer des fragments de l'arithmétique avec des propriétés de décidabilité et une complexité intéressantes.

La logique de différence (*Difference Logic*, DL) est le fragment de l'arithmétique qui gère des contraintes du type $x - y \leq c$, où x et y sont des variables et c est une constante numérique ; x et y peuvent être des variables entières ou rationnelles. Beaucoup de problèmes utilisent ce fragment uniquement. Dans de nombreux cas, même si l'on sort de ce fragment très restreint, la plupart des contraintes restent dans ce cadre. La SMT-LIB [59] contient de nombreux exemples de problèmes exprimés dans ce fragment restreint.

La logique de différence est bien étudiée et elle peut être modélisée complètement en utilisant la théorie des graphes. De ce fait, les solveurs SMT utilisant cela peuvent bénéficier des algorithmes efficaces de la théorie des graphes pour obtenir de bonnes performances sur ce langage.

Propriétés et représentation sous forme de graphes

Classiquement, les problèmes de logique de différence traitent uniquement des contraintes de la forme $x - y \leq c$. Une telle contrainte peut être représentée sous forme de graphe comme étant une arête de y vers x avec un coût (ou poids) c (voir Figure 1.4). Cela peut être compris comme : x doit être au plus $y + c$.

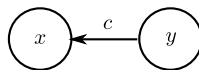


FIG. 1.4: Représentation de la contrainte $x - y \leq c$ sous forme de graphe.

Il existe de nombreuses autres contraintes pouvant être facilement traduites et intégrées dans ce modèle. La Table 1.2 donne certaines contraintes courantes qui peuvent facilement être traduites en DL.

Contrainte	Traduction
$x - y \geq c$	$y - x \leq -c$
$x - y = c$	$x - y \leq c$ et $y - x \leq -c$
$x \leq y$	$x - y \leq 0$
$x \geq y$	$y - x \leq 0$
$x = y$	$x - y \leq 0$ et $y - x \leq 0$
$x \leq c$	$x - v_0 \leq c$, où v_0 est une variable inédite unique avec la valeur 0
$x \geq c$	$x - v_0 \geq c$, où v_0 est une variable inédite unique avec la valeur 0

TAB. 1.2: Table de contraintes

Les inégalités strictes peuvent aussi être gérées avec des changements mineurs à l'algorithme. Les contraintes comme $x + y < c$ peuvent être réécrites $x + y \leq (c - \delta)$; cela implique de changer la représentation des nombres pour être capable de réaliser les opérations avec δ [31].

Une fois le graphe construit pour l'ensemble des contraintes de la logique de différence, on peut remarquer plusieurs choses.

Dépendance Si deux variables x et y ne dépendent pas l'une de l'autre, il n'y aura aucun chemin de x vers y , ou de y vers x .

Contrainte la plus forte La contrainte la plus forte est associée à une paire de variables. Si $y - x \leq c_1$ est la contrainte la plus forte associée à y et x , cela signifie qu'on ne peut extraire aucune autre contrainte de la forme $y - x \leq c_2$ où $c_2 < c_1$.

S'il existe un chemin de x à y , le chemin le plus court len de x à y donne la contrainte la plus forte ($y - x \leq len$) qu'il est possible de déduire des contraintes originales. Notons cependant que le chemin le plus court est relatif aux poids des arêtes de la source à la destination, et non du nombre d'arêtes qu'il contient.

Insatisfaisabilité Un ensemble de contraintes est insatisfaisable s'il est possible de trouver un sous-ensemble en contradiction, comme $x - y \leq -1 \wedge x - y \geq 2$. Si il n'existe aucune combinaison de contraintes pouvant rendre le problème inconsistant, le problème est satisfaisable. On peut toujours trouver un ensemble inconsistant si il existe un cycle négatif dans le graphe. En combinant linéairement les contraintes liées aux arêtes du cycle négatif, on obtient la contradiction.

Égalité entre variables Dans un problème satisfaisable, deux variables x et y sont égales si et seulement si le chemin le plus court de x à y a un

coût nul et le chemin le plus court de y à x a aussi un coût nul. L'explication peut être reconstruite des contraintes étiquetant les arêtes des chemins.

1.6.1 Procédure de décision pour la logique de différence

Test de satisfiabilité

La première fonctionnalité requise pour une procédure de décision est d'être capable de déterminer si un ensemble donné de contraintes est satisfaisable ou non. Un ensemble de contraintes est satisfaisable si et seulement si le graphe construit à partir des contraintes a un cycle négatif.

Notre algorithme est incrémental ; il est exécuté pour chaque nouvelle contrainte. Ceci peut être effectué jusqu'à $|E|$ fois, en stoppant le processus dès qu'un conflit est trouvé. L'idée de l'algorithme est de chercher dans le graphe les noeuds dont la distance change à cause de l'ajout de l'arête. La distance est la longueur du plus court chemin à partir d'un noeud arbitraire.

Pour ceci et pour la simplicité de l'algorithme, un noeud artificiel peut être créé, et être le noeud de référence. Il sera connecté à tous les autres noeuds avec une arête unidirectionnelle de coût 0. Comme aucun noeud n'aura une arête vers ce noeud arbitraire, ce noeud ne pourra jamais intervenir dans un conflit. Ainsi, le système original sera inconsistant si et seulement si le système intégrant ce noeud de référence l'est.

D'abord, l'algorithme commence par vérifier si la nouvelle arête e diminue la distance au noeud de destination. Si oui, la recherche commence ; le noeud dans le tas qui a sa distance diminuée le plus est pris en premier. On suppose que, quand un noeud v a sa distance modifiée, ses voisins vont aussi avoir leur distance modifiée d'au plus la modification qu'a subie v . Ainsi, quand un noeud choisi a sa distance modifiée, cela est fait par le chemin qui diminue le plus la distance. Donc, chaque noeud doit avoir sa distance améliorée au plus une fois. L'exception est justement lorsqu'il y a un cycle négatif. Un cycle négatif apparaît si et seulement si la distance de l'origine de e diminue.

Construction d'un ensemble de conflit

Quand l'insatisfaisabilité est détectée, il est nécessaire de générer une explication du conflit. Il s'agit en fait de l'ensemble des contraintes qui interviennent dans le conflit. Un ensemble de conflit minimal est obtenu en collectant les contraintes étiquetant les arêtes du cycle négatif.

Génération d'égalité

Dans notre modélisation par graphe, deux variables x et y sont égales si et seulement si les longueurs des chemins les plus courts entre x et y , et entre y et x sont 0. Un algorithme pour trouver les égalités, basé sur [42], procède

comme suit. Un graphe G' est construit, en collectant toutes les arêtes de G qui ont un *slack* nul. Une arête e a un $slack(e)$ nul si e est dans le chemin le plus court entre le noeud de référence et la destination de e . L'étape suivante de l'algorithme est de chercher les composantes fortement connexes dans G' . Deux noeuds d'une composante fortement connexe de G' sont dans un cycle de coût 0 et le chemin entre ces noeuds est aussi le chemin le plus court dans le graphe d'origine G . Il s'agit de la première condition pour que les variables étiquetant deux noeuds soient égales. Ensuite, chaque paire de noeuds, dans chaque composante fortement connexe sont potentiellement des candidats à l'égalité. Pour vérifier si deux variables v_1 et v_k sont égales sans avoir à calculer le chemin le plus court entre chaque paire de variables, on peut utiliser l'information de distance par rapport à un noeud de référence collectée ci-dessus. Deux variables v_1 et v_2 sont égales si et seulement si elles ont la même distance.

Explication d'une égalité

Si une égalité entre deux variables est générée, les prémisses de cette égalité sont les contraintes qui permettent de la déduire. Deux variables u et v sont égales si la longueur du chemin le plus court de u à v et de v à u sont 0. Pour reconstruire ensuite l'ensemble des prémisses, il suffit d'extraire les contraintes associées à chaque arête dans les deux chemins les plus courts. Cela peut être réalisé par une recherche en largeur d'abord.

Génération d'une égalité de modèle

La façon la plus simple d'obtenir un modèle hors du graphe calculé est d'utiliser l'information de distance déjà présente dans le graphe. Aucun coût supplémentaire n'est associé au calcul du modèle. Les égalités de modèle peuvent être déduites simplement des variables qui ont les mêmes valeurs.

Une fois générées, les égalités de modèle sont propagées, et, si aucune autre procédure de décision n'est en conflit avec ces égalités, le processus termine. Autrement, le modèle sera modifié, par l'envoi d'une diségalité. Les diségalités ne peuvent être incorporées telles quelles dans le graphe ; elles seront traitées de façon séparée.

La façon la plus simple de traiter des diségalités est de regarder la consistance par rapport au modèle. Si deux variables ont la même valeur mais qu'il existe une diségalité entre elle, la procédure de décision génère alors un lemme qui permettra d'intégrer indirectement l'inégalité dans le graphe, et ainsi de modifier le modèle.

Diminuer le nombre d'égalités de modèle

Il est possible, avec un peu de processing supplémentaire, de générer de meilleurs modèles. Un modèle est meilleur s'il génère moins d'égalités

de modèle. Il est préférable de générer moins d'égalités de modèle, car on diminue ainsi la probabilité d'avoir un conflit avec une autre procédure de décision. L'effet global est de diminuer le temps nécessaire à corriger des modèles.

Une variable sans borne inférieure ou supérieure peut avoir sa valeur dans un ensemble infini. En d'autres termes, il n'est même pas nécessaire de construire des égalités de modèles faisant intervenir ces variables.

Un autre raffinement important peut être obtenu en considérant les composantes fortement connexes. Des variables dans différentes composantes peuvent avoir leur valeur dans des intervalles différents. La procédure de génération de modèle peut elle-même se limiter à générer des égalités de modèle seulement entre les variables de la même composante fortement connexe.

Propagation de théorie

Supposons que $x - y \leq c$ est le littéral à vérifier. Si, dans le graphe, il existe un chemin de y à x de longueur c' , où $c' \leq c$, alors $x - y \leq c$ est une conséquence de l'ensemble de contraintes. Cela peut être réalisé grâce à un algorithme de plus court chemin avec source unique, comme l'algorithme de Dijkstra [29, 19], qui peut être implémenté en temps $O(|E| + |V| \log |V|)$.

Si il existe de nombreux littéraux qui peuvent être impliqués, cet algorithme de plus court chemin avec source unique sera utilisé un grand nombre de fois. Dans ce cas, il est peut-être préférable d'utiliser un algorithme de plus court chemin pour toute paire, comme l'algorithme de Floyd-Warshall [71, 19], qui peut être implémenté en $O(|V|^3)$.

1.7 Décider l'arithmétique linéaire

Il existe deux familles principales d'algorithmes pour l'arithmétique linéaire dans les solveurs SMT actuels. Une est basée sur l'élimination de *Fourier-Motzkin* [22], l'autre sur la méthode du *simplexe* [20]. Dans ce chapitre, nous nous intéressons au simplexe, à la base des algorithmes les plus efficaces en pratique [30, 61, 28, 51, 2].

1.7.1 La méthode du simplexe

La méthode du simplexe a été conçue en 1947 par George Dantzig et est une des méthodes les plus populaires pour la programmation linéaire. L'algorithme doit son nom au concept de simplexe, qui est la généralisation de la notion de triangle ou de tétraèdre à un espace de dimension arbitraire.

Habituellement, la méthode du simplexe résout des problèmes de programmation linéaire, c'est-à-dire des problèmes de maximisation (ou de minimisation) de fonction linéaire, restreinte par un ensemble de contraintes.

Par exemple :

$$\text{Maximiser : } F = 3x + 2y + z$$

$$\text{Sous : } 2x + y \leq 18$$

$$2x + 3y \leq 42$$

$$3x + y + 3z \leq 24$$

$$x, y, z \geq 0$$

$$x, y, z \in \mathbb{Q}$$

D'un point de vue géométrique, chaque inégalité réalise une coupure dans l'hyperespace. Aux intersections, il y a les arêtes et les sommets composent l'objet géométrique convexe, de polytope en n dimensions, où n est le nombre de variables. L'algorithme du simplexe commence à un sommet dans la région accessible (délimitée par les contraintes) et se déplace le long des arêtes de façon à augmenter la fonction à maximiser. Quand un maximum local est atteint, par convexité, il s'agit aussi d'un maximum global, ainsi l'algorithme s'arrête. La Figure 1.5 présente un exemple en trois dimensions (trois variables).

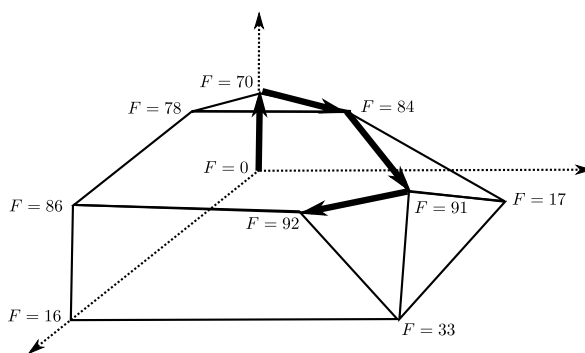


FIG. 1.5: En commençant d'un point dans la région accessible, le maximum de F est atteint en suivant les sommets tout en faisant croître la valeur de F .

Considérons maintenant le problème de la programmation linéaire avec deux variables et trois contraintes. On veut maximiser la fonction Z (c'est-à-dire trouver la solution optimale), sous trois contraintes. En plus, la méthode du simplexe impose des restrictions sur les valeurs des variables : $x, y \geq 0$.

$$\text{Maximiser : } Z = 2x + 3y$$

$$\text{Sous : } -x + y \leq 5$$

$$x + 3y \leq 35$$

$$x \leq 20$$

$$x, y \geq 0$$

D'abord, le problème est transformé en un système d'équations linéaires. Ensuite, un tableau est construit à partir de ce système d'équations

On définit deux classes de variables : les variables basiques et non basiques. Chaque équation dans le tableau possède exactement une variable basique et cette variable basique intervient dans uniquement cette équation. Les variables non basiques n'ont pas cette restriction. Intuitivement, les variables basiques sont définies par une combinaison linéaire des variables non basiques.

À tout instant, on a une solution basique associée au tableau. Dans une solution basique, toutes les variables non basiques ont une valeur nulle, et la valeur des variables basiques est obtenue en divisant la valeur dans la colonne la plus à droite par la valeur (non nulle) dans la colonne de la variable, soit, dans notre cas, toujours 1. La solution basique du tableau initial est présentée Figure 1.6.

	Z	x	y	s ₁	s ₂	s ₃	
s ₁	0	-1	1	1	0	0	5
s ₂	0	1	3	0	1	0	35
s ₃	0	1	0	0	0	1	20
Z	1	-2	-3	0	0	0	0

\implies
 \implies
 \implies
 \implies

$x, y = 0$
 $s_1 = 5/1 = 5$
 $s_2 = 35/1 = 35$
 $s_3 = 20/1 = 20$
 $Z = 0/1 = 0$

FIG. 1.6: Solution basique du tableau initial.

L'étape suivante est de sélectionner une variable qui est amenée à devenir basique. On examine les nombres dans la ligne inférieure, qui correspond à la fonction à maximiser, et on sélectionne le nombre négatif le plus petit (soit le plus grand en valeur absolue). Si il n'existe aucun nombre négatif dans la ligne inférieure, la solution optimale a été atteinte et l'algorithme s'arrête.

Il faut aussi sélectionner la variable qui quittera la base. Il s'agit de la variable qui limite le plus le changement de la valeur de la variable qui rentre dans la base, car on ne peut violer des contraintes. On calcule le ratio entre les nombres positifs dans la colonne de la variable entrante par le nombre dans la même ligne dans la colonne la plus à droite. La variable avec le plus

petit ratio est la variable qui restreint le plus le changement de valeur de la variable entrante, et c'est elle qui quittera la base. S'il n'y a aucune valeur positive, la solution est non bornée et l'algorithme termine.

L'étape suivante est le pivot. Des combinaisons linéaires pour exprimer les équations du tableau en termes des nouvelles variables basiques sont réalisées. Le résultat est donné Figure 1.7. La variable y est entrée dans la base et la variable s_1 l'a quitté. Les équations ont été réécrites de façon à ce que y apparaisse seulement dans une contrainte. La valeur de la fonction de maximisation Z est 15 dans la configuration présente. Cependant, il y a toujours la possibilité d'améliorer cette valeur, car il existe un nombre négatif dans la ligne du bas. Les étapes précédentes (depuis la sélection d'une variable qui quitte la base) sont répétées jusqu'à ce qu'aucun nombre dans la ligne du bas ne soit négatif.

	Z	x	y	s_1	s_2	s_3	
y	0	-1	1	1	0	0	5
s_2	0	4	0	-3	1	0	20
s_3	0	1	0	0	0	1	20
Z	1	-5	0	3	0	0	15

Leaving variable
↓

↑
Entering variable

New basic variable ←

$R'_1 = R_1$
 $R'_2 = R_2 - 3R_1$
 $R'_3 = R_3$
 $R'_4 = R_4 + 3R_1$

FIG. 1.7: Le résultat du pivot. La variable basique s_1 est remplacée par y et des combinaisons linéaires sont utilisées pour éliminer y des équations.

On peut aussi considérer le problème géométriquement (Figure 1.8). En associant les axes aux variables x et y , on obtient une représentation en deux dimensions du précédent exemple.

La région mise en évidence et délimitée par les contraintes du problème, est la région accessible. On commence au sommet ($x = 0, y = 0$) et on se déplace vers un autre sommet chaque fois qu'on peut augmenter la fonction à maximiser. La fonction de maximisation peut aussi être vue sur la figure. La flèche pointe dans la direction de maximisation.

1.7.2 Décider la satisfaisabilité incrémentalement

L'objectif est d'adapter la méthode du simplexe pour construire une procédure de décision pour l'arithmétique linéaire. La première difficulté est de lever les restrictions sur la forme des contraintes dans le simplexe primal. Notre procédure doit accepter des variables non bornées initiale-

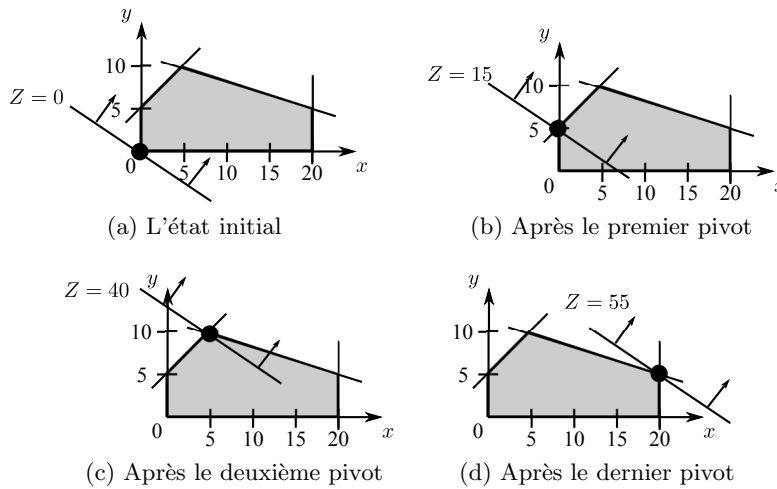


FIG. 1.8: Représentation géométrique

ment (contrairement à des variables positives uniquement), accepter des équations, négation d'équations, et inégalités. La deuxième difficulté est de construire une procédure incrémentale.

Notre problème est par ailleurs plus simple que le problème pour lequel le simplexe est conçu : nous ne demandons pas de solution optimale. Une solution suffit.

La première étape de l'algorithme est de vérifier s'il y a variables inédites dans la contrainte. Pour chaque nouvelle variable, les bornes inférieures et supérieures seront initialisées respectivement à $-\infty$ et ∞ . La valeur initiale de ces variables sera aussi fixée à une valeur arbitraire, 0.

On conserve l'idée de variables basiques et non basiques. Une variable basique apparaît dans une contrainte seulement, tandis qu'une variable non basique peut apparaître dans plusieurs. Les variables nouvellement créées sont non basiques initialement.

La différence principale avec le simplexe primal est que les variables non basiques peuvent avoir une valeur non nulle. Comme il n'existe pas de restriction sur la borne inférieure des variables, les variables peuvent avoir une valeur inférieure à 0. Une variable basique a toujours sa valeur parfaitement déterminée par la valeur des variables non basiques, mais, comme la valeur des variables non basique peut être non nulle, le calcul n'est pas aussi direct que dans le simplexe primal.

L'étape suivante est de normaliser la contrainte nouvellement ajoutée. Ceci est réalisé en remplaçant les variables basiques par les expressions qui les définissent, et qui contiennent des variables non basiques uniquement.

La nouvelle contrainte est alors considérée comme la fonction objectif, et l'algorithme essaie de la satisfaire, tout en gardant les autres contraintes valides. Ce n'est pas très différent du simplexe primal hors mis quelques

détails. La nouvelle contrainte est enfin intégrée à l'ensemble des anciennes contraintes. La façon dont elle est intégrée dépend de s'il s'agit d'une égalité, une négation d'égalité, ou une inégalité.

Si la contrainte est une négation d'égalité, elle est juste sauvegardée pour utilisation à la fin du processus. Les négations d'égalité ne peuvent être intégrées directement dans le simplexe, elles ne seront donc utilisées que plus tard en regard de la solution trouvée.

Si la contrainte est une inégalité, une nouvelle variable slack est créée pour permettre de transformer cette inégalité en égalité. Une valeur et une borne est donnée à cette nouvelle variable, et elle est mise dans l'ensemble des variables basiques. La valeur courante de cette variable est obtenue par différence entre l'évaluation des variables et le terme constant.

Si la contrainte est déjà une égalité, aucune variable slack n'est créée. Une variable est choisie pour être basique, et toutes les contraintes sont normalisées pour éliminer cette nouvelle variable basique, si nécessaire. Il existe cependant deux cas particuliers. Premièrement si la contrainte ne contient qu'une variable, les bornes de cette variable sont ajustées en fonction de la contrainte. Deuxièmement, si la contrainte ne contient aucune variable, elle est ignorée si elle est satisfaite.

L'algorithme appliqué sur l'ensemble de contraintes qui suit

$$\begin{aligned} -x + y &\leq 5 \\ x + 3y &\geq 35 \\ x &\geq 0 \\ y &\leq 5 \\ 3x + 3y &= 120 \end{aligned}$$

donne le tableau présenté Figure 1.9. La Figure 1.10 donne une représentation géométrique des différentes étapes.

	x	y	s_1	s_2	
y	0	6	0	-3	-15
s_1	0	0	-3	-3	-150
x	2	0	0	1	85

Bounds			
var	lower	value	upper
x	0	42.5	∞
y	$-\infty$	-2.5	5
s_1	0	50	∞
s_2	0	0	∞

FIG. 1.9: Le tableau final, après ajout des contraintes $-x+y \leq 5$, $x+3y \geq 35$, $x \geq 0$, $y \leq 5$ et $3x + 3y = 120$.

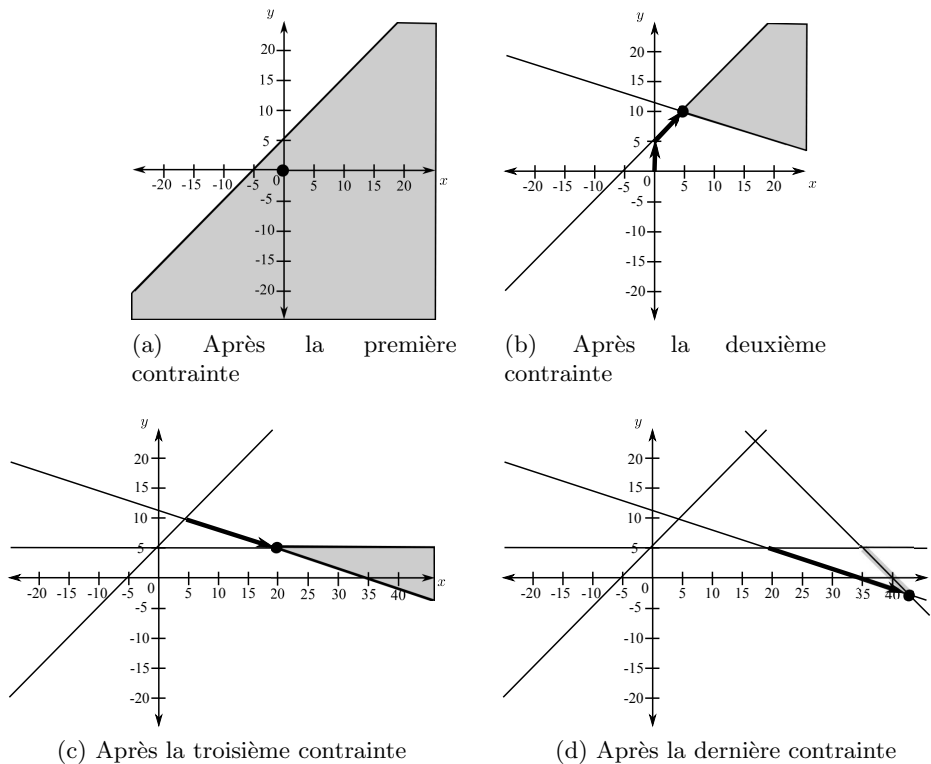


FIG. 1.10: La représentation géométrique après l'ajout des contraintes $-x + y \leq 5$, $x + 3y \geq 35$, $x \geq 0$, $y \leq 5$ et $3x + 3y = 120$.

1.7.3 Génération de l'ensemble de conflit

Le système est insatisfaisable si la nouvelle contrainte ne peut être satisfaite. Les variables sont alors directement contraintes par leurs bornes. Dans l'explication du conflit, il faut rendre un sous-ensemble des contraintes originales.

Premièrement, toutes les combinaisons linéaires (pendant la normalisation et les pivots) des contraintes originales sont sauvegardées par l'algorithme. Une expression linéaire donnant l'origine de la contrainte est gardée à côté de chaque contrainte. Deuxièmement, l'ensemble de conflit doit donner la raison des bornes des variables qui ne peuvent être modifiées. L'ensemble de conflit est alors juste l'ensemble de toutes ces contraintes. Comme l'information nécessaire à l'obtention de ces contraintes est maintenue à jour à chaque instant, la génération de conflit est très simple.

1.7.4 Backtracking

Un des aspects de notre algorithme est qu'il n'est pas nécessaire d'être à un sommet. Cette propriété simplifie grandement le backtracking.

Le backtracking doit ramener la procédure de décision dans un état correspondant à celui avant l'ajout de certaines contraintes. Dans notre cas, le backtracking ne restaurera pas exactement le même état, mais un état équivalent.

Si la procédure a atteint un état insatisfaisable juste à l'ajout de la contrainte précédente, le backtracking consiste simplement à retirer cette dernière contrainte. Si par contre la contrainte la plus récemment ajoutée a préservé la satisfaisabilité, la retirer n'invalide pas la solution. Dans les deux cas, il suffit donc de retirer la contrainte la plus récemment ajoutée. Il faut cependant veiller à retirer toute trace de cette contrainte, car il n'est pas impossible qu'elle soit utilisée dans une combinaison linéaire avec plusieurs autres contraintes. Les autres tâches liées au backtracking sont triviales, par exemple il s'agit de restaurer les valeurs de bornes sur des variables, ou d'éliminer des variables créées pour la contrainte.

1.7.5 Génération d'égalité

Les stratégies complètes pour générer toutes les égalités entre variables sont coûteuses. Mais il existe des solutions pour trouver certaines des égalités très rapidement. Par exemple, si deux variables ont la même valeur, et que cette valeur est totalement contrainte, l'égalité des deux variables est conséquence logique des contraintes.

Une heuristique un peu moins simple consiste à comparer les équations. Si deux variables basiques sont exprimées de la même façon en fonction des variables non basiques, elles sont égales. Par exemple, il est possible de déduire que $x = y$ de

$$x = 3z - 4w + 5$$

$$y = 3z - 4w + 5$$

Pour générer de manière complète toutes les égalités, il suffit d'utiliser le simplexe. Si, on atteint un état inconsistant en ajoutant $x < y$ d'une part et en ajoutant $x > y$ d'autre part, on sait que l'état courant implique $x = y$. Par contre, ce processus est coûteux.

1.7.6 Génération d'égalités de modèle

La génération d'égalités de modèle est triviale dans notre contexte. Il suffit de regarder les valeurs associées aux variables et de créer des égalités de modèle entre ces variables. L'algorithme est tel que les valeurs des variables seront généralement dispersées (contrairement à ce qui était observé pour la procédure pour la logique de différence). La procédure générera donc généralement un nombre assez faible d'égalités de modèle.

1.7.7 Diségalités et inégalités strictes

Les négations d'égalité ne peuvent pas être simplement intégrée au tableau. Elles sont donc gardées en mémoire et vérifiées plus tard. La procédure de décision pour l'arithmétique linéaire gère alors ces diségalités de la même façon que la procédure pour la logique de différence. La procédure de décision inspecte ces inégalités et génère les lemmes pour modifier le modèle courant si il y a une inconsistance entre le modèle et les diségalités.

Le inégalités strictes sont gérées de la même façon que pour la logique de différence. On remplace les contraintes du type $x + y < c$ par $x + y \leq (c - \delta)$, tout en modifiant la représentation des nombres de façon à gérer correctement les opérations avec δ .

1.7.8 Variables entières

La procédure de décision ci-dessus est complète pour les réels, et peut être intégrée dans un solveur SMT. Cependant, elle n'est plus complète en présence de variables entières.

Pour gérer les entiers, on ajoute aux algorithmes de simplexe des techniques comme le branch-and-cut [45], qui est un mélange de branch-and-bound avec cutting planes [36]. L'algorithme sur les réels est utilisé, et, chaque fois que la solution finale donne une valeur réelle à une variable entière, les techniques précitées sont utilisés pour éliminer ces solutions réelles et obtenir une solution entière.

L'idée derrière la technique du cutting planes est de générer une contrainte qui élimine la solution réelle courante, et un ensemble de solutions réelles autour, tout en préservant les solutions entières. Tant que la solution trouvée donne une valeur réelle à une variable entière, des coupures (cuts) sont générées sous la forme de nouvelles contraintes.

La technique du branch-and-bound crée une étude de cas aux variables entières avec valeurs réelles, en invalidant la solution actuelle. Par exemple, si x est une variable entière, et que sa valeur est 1.5, la technique du branch-and-bound introduira une analyse par cas, un sera $x \leq 1$ et l'autre $x \geq 2$. Si un des deux scénarios permet d'obtenir une solution entière, la solution est aussi solution du problème original.

Si cette technique du branch-and-bound est simple, la terminaison pose problème si les variables ne sont pas bornées. Elle ne suffit donc pas seule. Par contre la technique du cutting plane est complète, mais il n'est pas simple de générer les coupures. Les stratégies modernes utilisent les deux techniques.

Une stratégie complète reste à définir dans le solveur veriT. La technique du branch-and-bound est implémentée, en plus de la vérification du plus grand commun diviseur. Ensemble, ces deux techniques permettent de trouver la plupart du temps un modèle acceptable ou l'inconsistance.

1.8 Conclusion

Nous avons présenté des méthodes pour construire une coopération entre procédures de décision. Ces méthodes ont été appliquées aux fragments arithmétiques de la logique de différence et de l'arithmétique linéaire. La plateforme de combinaison est applicable à d'autres théories.

Après avoir présenté les solveurs SAT et solveurs SMT, montré comment des procédures de décision pour différentes théories peuvent être combinées au sein d'un solveur SMT, détaillé les fonctionnalités nécessaires pour construire des solveurs SMT efficaces, nous avons détaillé deux procédures de décision, une pour la logique de différence, et l'autre pour l'arithmétique linéaire.

La grande partie du travail présenté ici a été implémenté dans notre solveur SMT, `veriT`, qui est distribué en open-source sous la licence BSD, et téléchargeable sur le site <http://www.verit-solver.org>. Le module arithmétique, qui intègre les deux procédures de décision (logique de différence et arithmétique linéaire) a été implémenté en C, et fait de l'ordre de 9000 lignes de code. Le code est disponible avec le solveur `veriT`.

En résumé, les contributions sont :

- Une extension de la plateforme de combinaison de Nelson et Oppen qui utilise les égalités de modèle.
- Une description de l'implémentation des égalités de modèle pour la logique de différence.
- Une implémentation libre de la logique de différence, utilisée au sein du solveur SMT `veriT`.
- Une version incrémentale du simplexe, utilisée pour vérifier la satisfaisabilité d'un ensemble de contrainte en arithmétique linéaire.
- Une implémentation libre de l'arithmétique linéaire basée sur le simplexe, utilisée au sein du solveur SMT `veriT`.

Part II

Extended thesis

Chapter 2

Introduction

The construction of software that works correctly is a major concern in our society. But building software totally free of failures is most of the times a hard task. Many techniques may be applied during software construction to make them work as well as possible. Good software engineering techniques in project construction and development can help to achieve this goal, but usually they are not enough. Testing is the most common way to verify that a program works as expected, but it might be hard (or even impossible) to test it exhaustively. Formal methods try to prove formally models of software. If the models are proved, we can be totally sure of the correctness with respect to the proven properties. But as any of the other techniques, they have their limitations.

It is not all sort of software that have the real need to be entirely correct. For many applications the extra effort to ensure that everything works perfectly is not even worth. That is because using techniques to prove the correctness may take a long time and they may also not be easy to use. Sometimes, software can be released when it seems to work correctly, and later, when a problem comes out, it is fixed and a new version is released. That is usually what happens and it is not a serious concern if these hypothetical problems do not cause a big loss when it comes to lives or money.

The real necessity of software that works correctly comes from critical applications. This is where software deals with human lives, money, credibility, etc. Software that controls airplanes, space rockets, air traffic, metro, trains, and so on, deals with lives. Making sure it is correct is essential. Systems that deal with money, like bank and commercial softwares, can spend an extra effort during the construction of the software to make sure they will not lose money later. Operating Systems need to have the credibility that they will work correctly, as they are the base for many other software. Bug free software may also be a good propaganda for companies that produce such software, as they may gain the trust of their users. As a last example, embedded systems or hardware, that once out in the market cannot be

changed, need extra attention too.

Testing is a mandatory phase when making software. It is a good starting point and may detect many simple and common mistakes as well as deeper problems such as wrong algorithms. A simple way of completely testing the system and be sure it works correctly is to test all possible input configurations and to see if they return the expected results. But testing every input configuration is only possible for very few special cases as the number of configurations usually grows exponentially with the size of the input. So testing them all would take unreasonable time even for relatively small inputs. Also, in some situations, testing all the cases is impossible as they might be infinite. Therefore, for most software, testing techniques must be applied, using only a fraction of the whole input set of configurations. It will probably detect some errors, but it will not be possible to know if a program is free of failures, as it cannot test all input configurations.

Formal methods come to help proving the correctness of software. There are many techniques that aim to prove that an implementation of an algorithm works, based on formal theories. But proving that any code on any programming language gives always the expected result is not possible, as there is no formal theory that handles all the aspects of programming languages, compilers, operational systems, computer architectures, etc. So, normally, important aspects of the software are translated to a model that can be then checked.

After that, the proof may be done with the help of tools. There are many classes of tools available and they can be interactive or automated. Examples are SAT-solvers, SMT-solvers, extended static checkers, proof assistants, model-checkers... which use will depend on the problem and the specification. In this thesis we will focus on the components of the SMT-solvers.

Moreover, formal methods are in constant evolution. Researchers are often creating new methodologies, model languages and more robust provers. Although currently not attractive to all, formal methods are well used in industry in many areas. They are also in expansion, as one of the main goals is to be suitable for more users and situations.

The aspects to be proven usually have many theories involved such as: theory of lists, arrays, functions, numbers, sets, etc. Making a tool that can infer about a combination of them is not an easy task. A common way of doing it is to have a framework that combines decision procedures that can infer about some single theories, e.g., the Nelson and Oppen combination framework [52]. A survey about decision procedures for many different theories was written by Kroening and Strichman [40].

SMT-solvers are among the most successful state-of-art solvers for verification problems. Most SMT-solvers use the Nelson and Oppen combination framework to handle different combination of theories. The SMT-solver also have internally a SAT-solver which is a powerful tool to solve Boolean prob-

lems. The use of SAT-solvers combined with decision procedures makes the SMT-solver an interesting tool to solve expressive verification formulas. But as we are going to see in this thesis, making all these components work together is not trivial.

One of the theories most commonly found in verification problems is arithmetic. Concluding for instance that $x + x = 1$ is satisfiable in the real field, but not when variables are integer, is a very simple example. Arithmetic is a very large theory that can be divided in a few different fragments or classes, as problems that involve arithmetic usually contain only a subset of the arithmetic functions and symbols.

An interesting fragment of arithmetic is *difference logic*. It is based on constraints of the form $x - y \leq c$, where x and y are variables and c is a numerical constant. Although very simple, difference logic can express important practical problems like timed systems, scheduling problems and paths in digital circuits (see for example [54]). Difference logic can be entirely modeled with graph theory. That allows the use of fast algorithms capable of solving large instances of problems.

Another important fragment in arithmetic is linear arithmetic. It is more expressive than difference logic. In this fragment we are allowed to use addition, subtraction and multiplication as we want, except for multiplication between variables. Another advantage is to use as many variables per constraint as necessary, instead of only two of difference logic. An example of a linear arithmetic constraint is: $x_1 + 2x_2 - 5x_3 \leq 5$. Algorithms to solve linear arithmetic are slower than the ones to solve difference logic, but they are still able to handle large size problems.

We are going to focus on the difference logic and linear arithmetic fragments in this thesis. We start by proposing a combination framework based on Nelson and Oppen that makes it easier and more efficient to solve problems involving a combination of many different theories, including difference logic and linear arithmetic. Then we will show how to build decision procedures for these theories that can be used in this framework and integrated in an SMT-solver.

2.1 The overview of the thesis

In Chapter 3, we present the *satisfiability problem* for propositional logic and the SAT-solvers that are used to solve this kind of problem. Then we introduce the SMT-solvers that are used to solve more expressive generalizations of this problem.

In Chapter 4, we motivate and explain how we can combine the decision procedures related to different theories. We show the difficulties of combining some theories, in which situations that happens and we propose an original way of combining theories.

In Chapter 5, we start to see the details of the decision procedures. We analyze the requirements to build decision procedures that can be used in SMT-solvers.

In Chapter 6, we show how to build a decision procedure for difference logic using an original algorithm. We additionally describe all the algorithms to fulfill the requirements of a complete decision procedure that works in the proposed framework inside an SMT-solver.

In Chapter 7, we present the elements necessary to build a decision procedure for linear arithmetic. We present an original algorithm based on the *simplex method* and the extras necessary to have a cooperative decision procedure.

2.2 The publications during the thesis

- Thomas Bouton, Diego Caminha B. de Oliveira, David Déharbe and Pascal Fontaine. **GridTPT: a distributed platform for Theorem Prover Testing**. In Boris Konev, Renate Schmidt and Stephan Schulz, editors, In Proc. Workshop on Practical Aspects of Automated Reasoning, 2010.

The development of SMT-solvers requires intensive testing and experiments. During the period of my thesis, many new techniques were constantly incorporated and tried. It is important to certify the solver continues correct and observe the impact in the efficiency. This paper describes a distributed platform used in the tests of our SMT-solver veriT.

- Diego Caminha B. de Oliveira, David Déharbe, and Pascal Fontaine. **Combining decision procedures by (model-)equality propagation**. Science of Computer Programming, 2010.

This is the extended journal version article where we describe an original technique for combining decision procedures based on *model-equalities*. Part of this paper will be explained in Chapter 4.

- Thomas Bouton, Diego Caminha B. de Oliveira, David Déharbe and Pascal Fontaine. **veriT: an open, trustable and efficient SMT-solver**. In Renate A. Schmidt, editor, In Proc. Conference on Automated Deduction (CADE). LNCS. SpringerVerlag, 2009.

In this article, we find a general description and some of the techniques of our SMT-solver veriT.

- Diego Caminha B. de Oliveira, David Déharbe, and Pascal Fontaine. **Combining decision procedures by (model-)equality propagation**. In Brazilian Symposium on Formal Methods (SBMF2008). Editora Gráfica da UFBA, EDUFBA, 2008.

In this article, we describe an original technique for combining decision procedures based on *model-equalities*. Part of this paper will be explained in Chapter 4.

Chapter 3

From SAT to SMT-solvers

In this chapter, we start by briefly presenting the Boolean satisfiability problem. Then we show how to solve this problem and how to solve more expressive generalizations of it.

SAT-solvers are used to solve the satisfiability problem. A SAT-solver is also a core element of an SMT-solver. Although there are no contributions to SAT-solvers in this thesis, they are very relevant to the work presented through the thesis because of the role a SAT-solver plays inside an SMT-solver. The SAT-solver constantly interacts with the decision procedures, and is fundamental for understanding how to build efficient decision procedures for SMT-solvers. We can find a significant amount of work on SAT-solvers that was done in the past few years. However, the following sections will focus on discussing about basic techniques for building modern SAT-solvers that are most relevant to the remaining of the thesis.

The last part of this chapter presents SMT-solvers. An SMT instance is a generalization of the Boolean satisfiability problem, where we can find a variety of theories in the formula. SMT formulas provides a much richer modeling language than satisfiability formulas. We present the basic architecture and how to model problems to be solved by SMT-solvers.

3.1 Introduction

SMT-solvers are built on top of SAT-solvers, and the later were created to solve satisfiability problems. The satisfiability problem, also known as SAT problem, is the one that given a Boolean¹ (or propositional) formula, determine if it is possible to give values to the variables in such a way that the formula evaluates to true, or in other words, create an assignment that makes the formula true.

¹Where variables can only have *true* or *false* values.

An example of satisfiability problem is, given the formula:

$$(v_1 \vee v_2 \vee v_3 \vee v_4) \wedge (v_1 \vee v_2 \vee \neg v_3) \wedge (v_1 \vee \neg v_2) \wedge (\neg v_1 \vee v_4) \quad (3.1)$$

determine if it is possible to create an assignment, setting *true* or *false* values to the variables v_1, v_2, v_3 and v_4 , making the formula true.

A brute-force way of solving this problem tries all the combinations of assigning *true* or *false* to the variables, checking the formula each time to evaluate if it is *true*. The complexity is $O(2^n)$, for n variables. In the case of Formula 3.1 there would be 16 evaluations.

The satisfiability problem was the first problem proved to be NP-complete, in 1971 by Cook [18]. Since then, many other problems have been proved to be NP-complete. Among these proofs, several were done by reduction from the satisfiability problem.

However, despite the exponential worst-case nature of the problem, a great advancement was accomplished in SAT solving. State-of-the-art SAT-solvers can solve instances with millions of variables and several millions of constraints. The practical success of SAT-solvers challenges the relevance of worst-case complexity normally taken in consideration. In fact, SAT-solvers can very often surprisingly quickly solve instances from several different domains.

Due to this great capacity of handling large instances, many problems from different domains are modeled to SAT problem for making use of the high performance SAT-solvers. One can find examples of use of SAT-solvers in software and hardware verification [12, 69], automatic test pattern generation [66, 39], scheduling [35], planning [38, 62], algebra [72], etc. SMT-solvers also constitute an example of successful use of SAT-solvers, using them as core components to solve problems in a more expressive language.

The translation to satisfiability problem, i.e., using propositional representation, usually leads to a substantial increase in problem representation. That happens because of the limitation of using propositional representation, as it is going to be shown in Section 3.3. However, SAT encoding is not an obstacle to modern SAT-solvers anymore. In fact, many problems are solved faster by a SAT-solver, after been encoded to SAT, than by a custom search engine running on the original representation of the problem.

3.2 SAT-solver

In the past few years, a lot of progress was observed in the development of SAT-solvers. Annual SAT competitions, like the *SAT competition* [10, 9] and the *SAT-Race* [57] have contributed to the development of many smart implementations of SAT-solvers. From a non complete list of SAT-solvers found in these competitions websites, we can cite as examples: Barcelogic,

clasp, glucose, Jerusat, kw_aig, LySAT, ManySAT, march_hi, MiniSat, PicoSAT, plingeling, PrecoSAT, rcl, SATzilla, TNM, Zchaff...

Currently, the most successful SAT-solvers are based on variants of the procedure introduced several decades ago, the DPLL algorithm. Davis and Putnam [24] came up with the basic idea of the procedure and a couple of years later Davis, Logemann and Loveland [23] presented it in the efficient top-down form which is the base of most SAT-solvers today.

3.2.1 The DPLL algorithm

The DPLL algorithm performs a backtrack search in the space of the partial truth assignments. Its key feature is efficient pruning of the search space based on falsified clauses. Algorithm 2 shows the basic recursive version of the DPLL algorithm.

```

input :  $F$ : CNF_Formula
input :  $p$ : Assignment
output: status: {(SAT, Assignment); UNSAT }

// Unit propagation
1 while  $F$  has unit clause  $u \wedge$  contains no empty clause do
2   |  $F := F|_u$ ;
3   |  $p := p \cup \{u\}$ ;
4 end

5 if  $F$  contains the empty clause then return UNSAT ;
6 if  $F$  has no clauses left then return (SAT,  $p$ );

// Branching and decision
7  $\ell :=$  a literal not assigned by  $p$  ;
8 status,  $\Gamma :=$  DPLL( $F|_\ell$ ,  $p \cup \{\ell\}$ );
9 if status = SAT then return status,  $\Gamma$  ;
10 return DPLL( $F|_{\neg\ell}$ ,  $p \cup \{\neg\ell\}$ );

```

Algorithm 2: Basic recursive DPLL.

Algorithm 2 runs on CNF² formulas. The idea is to repeatedly select an unassigned literal ℓ in the input formula F and recursively search for a satisfying assignment $F|_\ell$ ³ (setting ℓ to *true*) and $F|_{\neg\ell}$ (setting ℓ to *false*). The step where ℓ is chosen (line 7) is called the *branch* step. The step where ℓ is set to *true* (line 8) or *false* (line 10) is called *decision* which is associated to a decision level equal to the depth of the Algorithm 2 recursion. The end of the recursive call, which takes F back to fewer assigned variables, is called *backtracking* step.

²CNF is for Conjunctive Normal Form, i.e., conjunction of disjunctions. Every formula can be converted in linear time to CNF, see e.g. [15].

³The notation $F|_\ell$ denotes the simplified formula obtained by replacing ℓ by *true* and $\neg\ell$ by *false*, later removing all clauses with at least one *true* literal, and deleting all occurrences of *false* literals from the remaining clauses.

A partial assignment p is maintained during the search. If $F|_p$ contains the empty clause, the corresponding clause of F from which it came is said to be *violated* by p . In the *unit propagation* step (lines 1 to 4), unit clauses are immediately set to *true* for efficiency reasons. *Pure literals* (those whose negation does not appear) may be set to *true* as a preprocessing step and, in some implementation, during the simplification after every branch.

Variations of this algorithm form the mostly used family of complete procedures for SAT solving. They are frequently implemented in an iterative way rather than recursively, resulting in a reduced memory usage. The drawback is the extra step necessary for unassigning variables when one backtracks. The naive way which requires to examine every clause is expensive, but modern techniques, such as two watched literals (presented later in this chapter), provides an excellent way of dealing with this.

3.2.2 DPLL, an example

Consider the Formula F in 3.2.

$$F : (\bar{a} \vee b \vee c) \wedge (a) \wedge (\bar{b} \vee c) \wedge (\bar{b} \vee \bar{c}) \wedge (b \vee c) \quad (3.2)$$

We want to know if there is a propositional assignment that makes F true. For this we simulate DPLL, as shown in Figure 3.1. In this example, for conciseness, overline is used on the literals to denote the negation, e.g., $\neg a \equiv \bar{a}$. In the figure, each clause is between parenthesis and in a different line. The boxes represent different states of the simulation of the algorithm.

- The run of the algorithm starts by detecting a unit clause, (a) , so a unit propagation is performed. The variable a is set to *true*, the changes are applied into the formula and then some simplifications are done. The simplifications in the formula are done over the *true* and *false* literals presented in the clauses. In this first assignment, the second clause, containing *true* is removed and the falsified proposition in the first clause is also removed.
- Following, with no unit clauses in the formula, a variable needs to be chosen in the branch step. Among the possible choices, the variable b is chosen and the algorithm decides to set the literal b value to *true*.
- A unit propagation can be performed. Both \bar{c} and c are candidates for unit propagation. The algorithm just choses the literal from the first unit clause, in this case the literal c . In either way, applying unit propagation leads to the empty clause (\emptyset) and therefore to an unsatisfiable assignment.
- The algorithm backtracks to a state where a decision was made, in this case when it decided b . Now it simply inverts the decision, setting b to false.

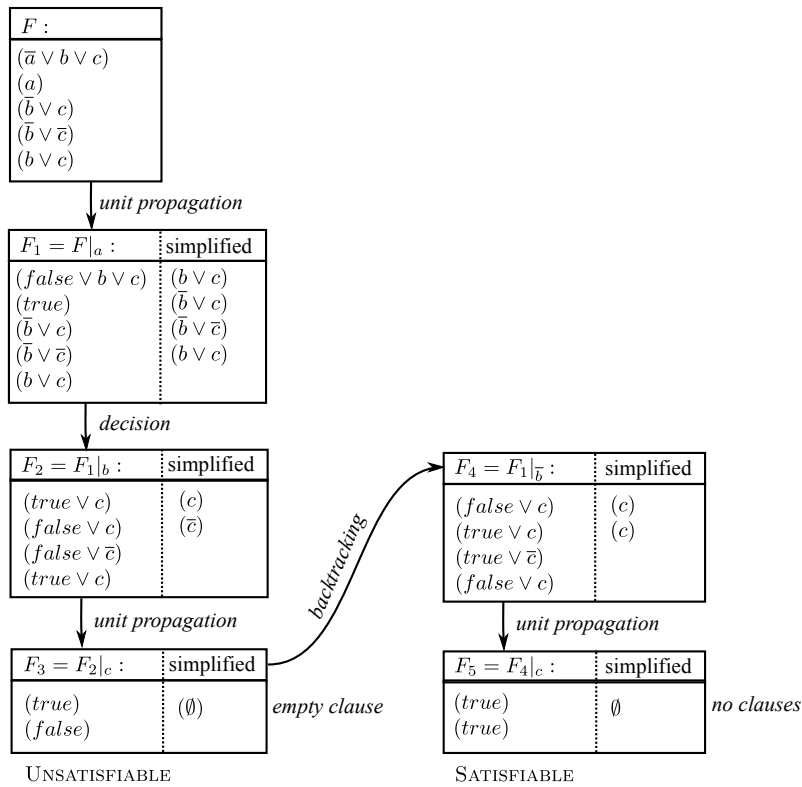


Figure 3.1: Execution of DPLL on example of Formula 3.2. Starting with the formula F , the algorithm applies, in successive steps, *unit propagation*, *decision* and *backtracking* to reach a propositionally true assignment, concluding that the formula satisfiable.

- Once again this generates unit clauses. The variable c is set to *true* and after simplification, no more clauses remain.
- With no more clauses, we reach a state where the algorithm detects a propositional satisfiable assignment. In Algorithm 2, the assignment is denoted by p . In Figure 3.1, the assignment can be reconstructed by looking at the decisions made. In this example it is $\{a, \bar{b}, c\}$, or, in other notation, $\{a = true, b = false, c = true\}$.

3.2.3 Modern techniques for DPLL based SAT-solvers

The efficiency of state of the art SAT-solvers relies heavily on several features that have been developed and tested in the last decade. Among these features are fast unit propagation using watched literals, learning mechanisms, deterministic and randomized restart strategies, constraint database

management for clause deletion, non chronological backtracking, and smart branching and decision heuristics.

Although these features are not essential for understanding the behavior and integration of a SAT-solver inside a SMT-solver, they are very important when considering building modern and efficient SAT-solvers. Following, we give a brief description of some of these features.

Two watched literals. The two watched literals scheme was developed by Moskewicz et al. [48] in their solver *zChaff*. It played a critical role in the success of SAT-solvers and is now a standard technique used by most SAT-solvers to find unit clauses and perform efficient unit propagation.

As the name suggests, the idea behind this scheme is to maintain and watch two initially unassigned literals for each non-unit clause. The key observation of this method is that as long as a clause has two unassigned literals, it cannot be involved in unit propagation. By doing a few low cost maintenance steps when one of these two variables have their value assigned, it is possible to simplify the search for unit clauses, increasing the solver efficiency specially when clauses in the formula are large. Another gain with this technique is backtracking. Since the invariant of the watched literals is maintained when unassigning literals, backtracking is done in constant time.

Branching and decision heuristics. Good heuristics to determine which variable to choose and which value to set is one of the features that vary the most from one SAT-solver to another. Different strategies can have a significant impact on the efficiency of the solver (see e.g. [63, 41] for surveys).

The strategies vary from randomly fixing the variables to moderately complex functions that take in consideration, for instance, how often a variable appears in conflicts and/or unsatisfied clauses, and how big are the clauses containing such a variable. In these more sophisticated strategies, variables are selected based on weights that are initially attributed to them and dynamically updated as the search progresses. Examples of decision heuristics are: maximum occurrence in clauses of minimum size [37]; dynamic largest individual sum [43]; variable state independent decaying sum [48]. New solvers like BerMin, Jerusat, MiniSat, and RSat employ further variations on this theme.

Clause learning. This technique also played a crucial role in the recent success of SAT-solvers and is an important improvement to the basic DPLL algorithm. The idea is to learn the causes of conflict by adding new clauses that will help pruning the search in a different part of the search space encountered later. There are many different schemes for conflict driven clause learning. Examples can be found in works of Marques-silva, Bayardo, Zhang [44, 7, 74].

Clause learning alone can be very limited by the explosion in the number of clauses that a SAT-solver may learn during the search. Its success is very connected to subsequent researches in lazy data structures and constraint database management strategies, for deletion of some previous learned clauses. Techniques for reducing the size of the conflict driven learned clauses also contribute to a more efficient learning process. Examples are conflict clause minimization introduced by *MiniSat* [64] and assignment stack shrinking introduced by *Jerusat* [50].

Backjumping. It is a technique used to reduce the search space and, therefore, increase the efficiency. While backtracking goes back in one level to change the value of the last decided variable and then continue the search, backjumping may go further. It may go back several levels of decision if it is possible to safely determine that there is no solution from that point on, independently of the values the unassigned variables may take. This generic technique was introduced by Stallman and Sussman [65] and now has variants in different solvers.

Restarts. Introduced by Gomes et al. [35], this technique allows clause learning algorithms to arbitrarily stop the search and restart their branching process from decision level zero. All learned clauses are kept and the algorithm may take a new, different branch. Gomes et al. showed that with a controlled randomized restart strategy it is possible to escape of *heavy-tailed* branches that very often the solvers face when running on hard problems. Combined with clause learning, this strategy improves the solvers by several orders of magnitude. Most of the current SAT-solvers apply aggressive restart strategies.

3.3 An example of modeling in SAT

Usually, there are several ways of modeling a problem in SAT. The performance of the solvers depends on the way the problem is modeled. In this section we give an example of modeling a classical NP-complete problem into a SAT encoding.

The problem is the *graph K -coloring*. It consists in determining if it is possible to assign colors to the nodes of a graph using at most K colors in such a way that no adjacent nodes have the same color. An example of a 3-colored graph is given in Figure 3.2. Observe that there is no 2-coloring of this graph.

In SAT problems, variables can only have *true* or *false* values. So when modeling, this is the first point to take in consideration. Another point is that formulas are usually given in CNF, i.e., conjunction of disjunctions, or in other words, a set of clauses. It happens very often that problems are

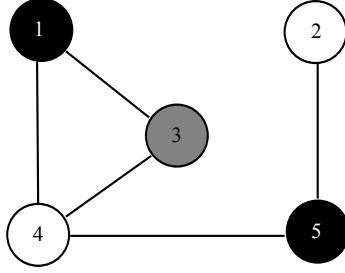


Figure 3.2: An example of a graph colored using $K = 3$ different colors: white, black and gray.

easily modeled using clauses, but if that is not the case, a conversion to CNF can be performed [15] and many solvers implement CNF transformation.

In this example, we start by defining the variables and then we add the constraints. It is modeled using clauses, directly in CNF, so no subsequent translation is required.

- Variables have the format x_{ij} meaning that the node i has color j . We do this for all the combination of N nodes and K colors, yielding $N \times K$ variables.
- Each node must have at least one color. So for each node i we create a clause specifying that: $(x_{i1} \vee x_{i2} \vee \dots \vee x_{iK})$, or,

$$\bigwedge_{i=1}^N (\bigvee_{j=1}^K x_{ij})$$

- Each node must have at most one color, i.e., for no node two colors are set *true*. So for each node i , we create a clause for each pair of colors: $(\neg x_{i1} \vee \neg x_{i2}) \wedge (\neg x_{i1} \vee \neg x_{i3}) \wedge \dots \wedge (\neg x_{i(K-1)} \vee \neg x_{iK})$, or,

$$\bigwedge_{i=1}^N \bigwedge_{j=1}^{K-1} \bigwedge_{\ell=j+1}^K (\neg x_{ij} \vee \neg x_{i\ell})$$

- No adjacent nodes may have the same color. So for each edge (i, j) from the set of edges E of the graph and for each color c , we create a clause: $(\neg x_{ic} \vee \neg x_{jc})$, or,

$$\bigwedge_{(i,j) \in E} \bigwedge_{c=1}^K (\neg x_{ic} \vee \neg x_{jc})$$

For a concrete example of encoding in SAT, we take the graph of Figure 3.2. In the case of $K = 3$ colors, the final encoding in SAT is:

$$\begin{aligned}
& (x_{11} \vee x_{12} \vee x_{13}) \wedge \\
& (x_{21} \vee x_{22} \vee x_{23}) \wedge \\
& (x_{31} \vee x_{32} \vee x_{33}) \wedge \\
& (x_{41} \vee x_{42} \vee x_{43}) \wedge \\
& (x_{51} \vee x_{52} \vee x_{53}) \wedge \\
& (\neg x_{11} \vee \neg x_{12}) \wedge (\neg x_{11} \vee \neg x_{13}) \wedge (\neg x_{12} \vee \neg x_{13}) \wedge \\
& (\neg x_{21} \vee \neg x_{22}) \wedge (\neg x_{21} \vee \neg x_{23}) \wedge (\neg x_{22} \vee \neg x_{23}) \wedge \\
& (\neg x_{31} \vee \neg x_{32}) \wedge (\neg x_{31} \vee \neg x_{33}) \wedge (\neg x_{32} \vee \neg x_{33}) \wedge \\
& (\neg x_{41} \vee \neg x_{42}) \wedge (\neg x_{41} \vee \neg x_{43}) \wedge (\neg x_{42} \vee \neg x_{43}) \wedge \\
& (\neg x_{51} \vee \neg x_{52}) \wedge (\neg x_{51} \vee \neg x_{53}) \wedge (\neg x_{52} \vee \neg x_{53}) \wedge \\
& (\neg x_{11} \vee \neg x_{31}) \wedge (\neg x_{12} \vee \neg x_{32}) \wedge (\neg x_{13} \vee \neg x_{33}) \wedge \\
& (\neg x_{11} \vee \neg x_{41}) \wedge (\neg x_{12} \vee \neg x_{42}) \wedge (\neg x_{13} \vee \neg x_{43}) \wedge \\
& (\neg x_{31} \vee \neg x_{41}) \wedge (\neg x_{32} \vee \neg x_{42}) \wedge (\neg x_{33} \vee \neg x_{43}) \wedge \\
& (\neg x_{41} \vee \neg x_{51}) \wedge (\neg x_{42} \vee \neg x_{52}) \wedge (\neg x_{43} \vee \neg x_{53}) \wedge \\
& (\neg x_{21} \vee \neg x_{51}) \wedge (\neg x_{22} \vee \neg x_{52}) \wedge (\neg x_{23} \vee \neg x_{53})
\end{aligned}$$

The formula is satisfiable, as it is related to the Figure 3.2 where we can see that there is a solution. If we give the formula to a SAT-solver, it will confirm that the formula is satisfiable. We can additionally ask for a model that will show which variables are *true* and which are *false*. With this information we can determine the color of each node, rebuild the graph with the colors, and do additional checking to ensure that there are no errors in the translations.

3.4 SMT-solver

We presented so far how to solve satisfiability problems, but we want to go one step further. We want to be able to solve Boolean problems also involving expressive theories, such as linear arithmetic, uninterpreted predicate and functions, and various data structures. In other words, we want to solve SMT⁴ problems.

Before the appearance of SMT-solvers, the entire theory inside a problem needed to be encoded in the propositional level if one desired to use a SAT-solver to tackle the problem. Now this is no longer necessary in most of

⁴SMT stands for Satisfiability Modulo Theories

the cases. SMT-solvers use the power of SAT-solvers in combinations with decision procedures capable of reasoning about different theories.

Many SMT-solvers have been developed in the last few years. Some SMT-solvers try to specialize in a few theories while others try to gather as many theories and combinations of theories as possible.

The annual SMT competition, the SMT-COMP [3, 1], also encourages the development of SMT-solvers. Additionally to our SMT-solver `veriT`, which implements most of the concepts presented in this thesis, from a non exhaustive list of SMT-solvers⁵, we can also cite as examples: Alt-Ergo, AProVE NIA, Barcelogic, Beaver, Boolector, `clsat`, CVC, MathSAT, MiniSmt, OpenSMT, `simplifyingSTP`, Sateen, Sonolar, Spear, STP #101, Sword, Yices, Z3.

The basic architecture of an SMT-solver can be seen in Figure 3.3. A SAT-solver is the core element. It is used to search for propositional satisfiable assignments. These assignments need to be verified to check if there are any theory inconsistency. In SMT-solvers based on the Nelson and Oppen combination framework (presented in Chapter 4), the assignment is sent to a module that will distribute each constraint to its respective decision procedure. The decision procedures check for inconsistencies, sharing some information with each other whenever there are several theories involved. If there are inconsistencies, lemmas are sent back to the SAT-solver, invalidating the current assignment and forcing it look for a new one. The SMT-solver stops when it finds an assignment with no inconsistencies at the theory level or when there are no more propositionally satisfiable assignments.

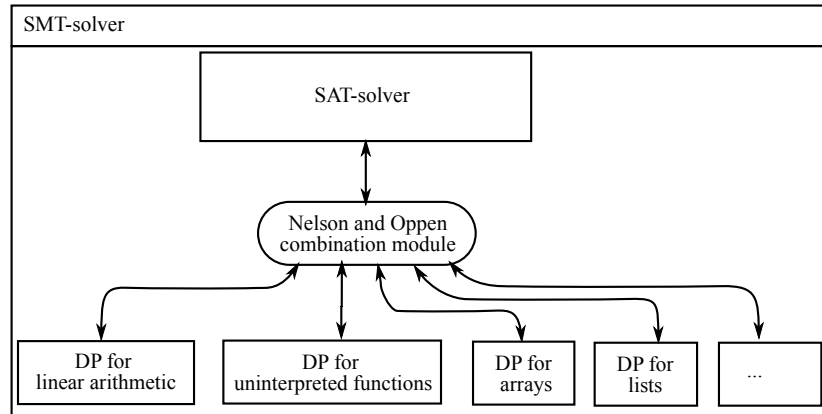


Figure 3.3: Basic architecture of an SMT-solver based on Nelson and Oppen framework and using several decision procedures (DP).

When there is only one theory involved, it is easy to integrate a decision

⁵Links and short description of the solvers can be found in the `smtcomp` web-page [1].

procedure with a SAT-solver to make a working SMT-solver. However, when there are two or more theories, the combination may not be trivial. We are going to see the details of how to combine theories in Chapter 4, and how to build decision procedures that can be integrated in an SMT-solver in Chapters 5, 6 and 7.

The SMT community makes available a benchmark library, SMT-LIB [59], where people can submit their own benchmarks or use the existing ones to develop the SMT-solvers. Benchmarks are in the standard language, also created by the community, SMT-LIB 1.2 [58] or SMT-LIB 2.0 [5]. Most of the SMT-solvers support one of these two languages.

In the next section, it is presented an example showing how to model a problem using SMT. We are not using an existing language, but the conversion to SMT-LIB 1.2 or 2.0 is simple.

3.5 An example of modeling in SMT

We want to model the same K -coloring problem of Section 3.3, but now with a more natural encoding that can be used by an SMT-solver. A natural way of encoding colors is to use numbers, representing each color by a number. Therefore, the integer linear arithmetic theory is used to model the 3-coloring problem of Figure 3.2.

Explaining step by step the encoding as it was done in the SAT case:

- For each node i , there is an integer variable x_i . The values the variables may have are between 1 and K (in our case $K = 3$ colors).
- Each node must have at least one color. In the arithmetic theory, variables always have one value, so we do not need to include any clause.
- Each node must have at most one color. In the arithmetic theory, variables are never assigned two values, so we do not need to include any clause.
- No adjacent nodes may have the same color. For each edge (i, j) from the set of edges E of the graph we add: $x_i \neq x_j$, or in a precise formalism,

$$\bigwedge_{(i,j) \in E} (x_i \neq x_j)$$

Modeling this problem with linear arithmetic results in a conjunctive set of literals in the theory of integers. In practice it means that a decision procedure is enough to solve it, and not the full SMT-solver. So to make the example more illustrative, we add one extra constraint to the problem.

We want also to know if all the colors have been used to solve the problem. To encode this extra constraint we do like the following:

- Each color $c \in K$ has at least one variable assigned to it, i.e.,

$$\bigwedge_{c=1}^K \bigvee_{i=1}^N x_i = c$$

The final encoding in SMT of the graph in the Figure 3.2, for $K = 3$, is:

$$\begin{aligned} x_1 &\geq 1 \wedge x_1 \leq 3 \wedge \\ x_2 &\geq 1 \wedge x_2 \leq 3 \wedge \\ x_3 &\geq 1 \wedge x_3 \leq 3 \wedge \\ x_4 &\geq 1 \wedge x_4 \leq 3 \wedge \\ x_5 &\geq 1 \wedge x_5 \leq 3 \wedge \end{aligned}$$

$$\begin{aligned} x_1 &\neq x_3 \wedge x_1 \neq x_4 \wedge \\ x_2 &\neq x_5 \wedge \\ x_3 &\neq x_4 \wedge \\ x_4 &\neq x_5 \wedge \end{aligned}$$

$$\begin{aligned} (x_1 = 1 \vee x_2 = 1 \vee x_3 = 1 \vee x_4 = 1 \vee x_5 = 1) \wedge \\ (x_1 = 2 \vee x_2 = 2 \vee x_3 = 2 \vee x_4 = 2 \vee x_5 = 2) \wedge \\ (x_1 = 3 \vee x_2 = 3 \vee x_3 = 3 \vee x_4 = 3 \vee x_5 = 3) \end{aligned}$$

The formula is satisfiable, as we have seen in Section 3.3. It is simpler to understand and shorter than the SAT formula. If we translate the formula to an existing input language and give it to an SMT-solver, it will check if the formula is satisfiable. SMT-solvers do not offer the same functionality, but there are many features available. In the case the result is satisfiable, we can ask for a model, for the assignment, the value of a variable, etc. In the case it is unsatisfiable, a proof of inconsistency can be obtained. With these extra informations that some SMT-solvers can offer, we can obtain the color in each node, rebuild the graph with the colors, obtain an unsatisfiable subgraph...

3.6 Conclusion

In this chapter, we presented an essential component of SMT-solvers, the SAT-solver. We showed the basic algorithm which most of the modern SAT-solvers are based on, and a short description of advanced techniques. Then, we presented an example of how to encode a problem in SAT.

The second part of the chapter we presented the SMT-solvers. We described its basic architecture and how it works. We ended with an example of how to encode in SMT.

In the next chapter we explain how we can combine the decision procedures related to different theories that are inside the SMT-solvers. We are going to see the difficulties of combining some theories, in which situations that happens and we also propose an original way of combining theories. There is also more examples where we can see step by step the interactions between the SAT-solver and the theory reasoners.

Chapter 4

Deciding a combination of theories

Formulas in the verification domain very often contain terms that come from different theories. Producing formulas using several theories is more natural than trying to encode everything using only a limited in expressiveness theory. Also, by using several theories, it is easy to express properties that otherwise would be very hard or impossible to encode. Examples of common theories that are frequently found in verification problems are arithmetic, uninterpreted functions, arrays, lists, sets and bit-vectors.

Verifying formulas containing symbols from a combination of theories is not straightforward. SMT-solvers employ techniques for combining the individual decision procedures and make them cooperate. Each decision procedure works on the subset of the formula referring to its theory. If one of them finds unsatisfiability, the problem is unsatisfiable. However, it is not necessarily satisfiable if no inconsistency is found by the decision procedures. A simple example with two decision procedures, one for arithmetic and another for uninterpreted functions is in $x = 0, y = 1 - 1, f(x) \neq f(y)$. The arithmetic decision procedure receives the constraints $x = 0$ and $y = 1 - 1$ and finds no inconsistency. The uninterpreted functions decision procedure receives $f(x) \neq f(y)$ and finds no inconsistency either. But, as we are going to see, they can deduce that this small problem is unsatisfiable if they share some information.

In this chapter we present techniques for deciding not just a single theory, but a combination of different ones. We start by presenting the *Nelson and Oppen combination framework* that works very well for some combinations of theories. Later we present an extension by introducing *model-equalities* that allows a better combination of a wider range of theories, including *integer arithmetic*.

4.1 The Nelson and Oppen combination framework

The Nelson and Oppen combination framework was first presented in [52, 53] as a way to decide by cooperating decision procedures. Two decades later, it was adopted by most SMT-solvers as an efficient method for combining some theories, and nowadays, it is still state-of-art.

Nelson and Oppen state that if theories \mathcal{T}_1 and \mathcal{T}_2 are disjoint¹ and stably infinite², the satisfiability of $\mathcal{T}_1 \cup \mathcal{T}_2$ can be determined by checking the satisfiability of $\mathcal{T}_1 \cup \mathcal{L}$ and $\mathcal{T}_2 \cup \mathcal{L}$ separately, where \mathcal{L} is a set of shared information. Therefore, if we have two decision procedures, one for \mathcal{T}_1 and another for \mathcal{T}_2 , the extra effort for checking the satisfiability of $\mathcal{T}_1 \cup \mathcal{T}_2$ remains in finding \mathcal{L} . This statement can be extended to more than two theories.

4.1.1 Equality generation and propagation

The simplest scenario is when all the theories involved are convex. A theory \mathcal{T} is convex when all the disjunctions of literals $l_1 \vee l_2 \vee \dots \vee l_n$ that can be deduced from it is a consequence of one of the literals l_i in the disjunction. Or more formally, $\mathcal{T} \models l_1 \vee l_2 \vee \dots \vee l_n$ implies $\mathcal{T} \models l_i$ for some $1 \leq i \leq n$.

Many theories are convex. *Real linear arithmetic*, *real difference logic*, *uninterpreted functions* and *lists* [56] are examples of convex theories. On the other side, *integer arithmetic*, *non-linear real arithmetic*, *sets* and *arrays* are examples of non-convex theories. Nelson and Oppen observe that the set of shared information \mathcal{L} can be precisely determined in the case of convex theories by repeatedly deducing and propagating equalities between shared variables³ until unsatisfiability is found or no more equalities can be deduced by any of the decision procedures.

This process works as follows. First, literals are sent to the decision procedures. Each decision procedure will receive the literals related to its theory. If necessary, constraints will be *purified*, by creating new variables, so that in the end no constraint will contain symbols from more than one theory. If unsatisfiability is found, the original set of literals is unsatisfiable and the process stops. Otherwise, the decision procedures will look for equalities that can be deduced from the current set of information and propagate these equalities to the other decision procedures. With these new constraints, the decision procedures may once again check for unsatisfiability and new equalities. The cycle continues until no more equality can be deduced by

¹Two theories are disjoint if no symbol appear in both theories, except for variables and the equality symbol (=).

²A theory is stably infinite if their satisfiable models have an infinite domain. Or more precisely, a theory is stably infinite if every quantifier-free formula in the theory that is satisfiable, is satisfiable by an interpretation with an infinite domain.

³Shared variables are the variables that appear in more than one theory.

any of the decision procedures or unsatisfiability is found.

As an illustrating example, consider the Formula 4.1. The formula contains information from three theories. There is linear arithmetic for reals with the relational operator \leq , the subtraction function $-$ and the number 0; the uninterpreted function f and predicate P ; and list functions car (that returns the first element of a list) and $cons$ (that adds a new element to the head of a list and returns the new list). Simulating the Nelson and Oppen combination framework, we want to prove that Formula 4.1 is unsatisfiable.

$$x \leq y \wedge y \leq car(cons(x, \ell)) \wedge P(f(x) - f(y)) \wedge \neg P(0) \quad (4.1)$$

First, the formula is purified, so that every constraint contains symbols from one theory only. This is done by using new variables to refer to terms of a constraint that are from another theory, and creating new constraints explaining what these variables mean. The resulting formula is equisatisfiable⁴. For example, there are terms from two different theories in the constraint $\neg P(0)$. The arithmetic term 0 is inside the uninterpreted predicate P . To purify it, we create a variable v_5 that will replace 0, getting $\neg P(v_5)$, and we create a new constraint $v_5 = 0$ that explains what v_5 means. Purifying the entire Formula 4.1 we get Formula 4.2.

$$\begin{aligned} x \leq y \wedge y \leq v_1 \wedge v_2 = v_3 - v_4 \wedge v_5 = 0 \\ \wedge P(v_2) \wedge \neg P(v_5) \wedge v_3 = f(x) \wedge v_4 = f(y) \\ \wedge v_1 = car(cons(x, \ell)) \end{aligned} \quad (4.2)$$

Formula 4.2 is a conjunction of literals. There is only one propositional assignment that makes the formula true, where all literals are set to *true*. In this case the SAT-solver does not play a relevant role.

The literals are dispatched to their respective decision procedures. There are three decision procedures: linear arithmetic for reals, uninterpreted functions and lists. They are identified in Figure 4.1 by \mathcal{T}_{LRA} -solver, \mathcal{T}_{UF} -solver and \mathcal{T}_L -solver, respectively. Following, there is a step by step explanation of the simulation.

State 1.0 - The decision procedures receive the constraints. All the decision procedures are executed and none of them can detect unsatisfiability. However, equalities can be deduced. $v_1 = x$ is detected ($cons$ adds the element x to the beginning of list ℓ and car returns the first element of the resulting list, i.e., x) by \mathcal{T}_L -solver and propagated to the other decision procedures.

State 1.1 - With a new constraint received, both \mathcal{T}_{LRA} -solver and \mathcal{T}_{UF} -solver can be executed again. The set of constraints is almost the same, except for

⁴Two formulas are equisatisfiable if the first formula is satisfiable whenever the second is and vice versa.

	\mathcal{T}_{LRA} -solver	\mathcal{T}_{UF} -solver	\mathcal{T}_L -solver
State 1.0	$x \leq y$ $y \leq v_1$ $v_2 = v_3 - v_4$ $v_5 = 0$	$P(v_2)$ $\neg P(v_5)$ $v_3 = f(x)$ $v_4 = f(y)$	$v_1 = \text{car}(\text{cons}(x, \ell))$ $v_1 = x$ (detected)
State 1.1	$v_1 = x$ (new) $x = y$ (detected)	$v_1 = x$ (new)	
State 1.2		$x = y$ (new) $v_3 = v_4$ (detected)	$x = y$ (new)
State 1.3	$v_3 = v_4$ (new) $v_2 = v_5$ (detected)		$v_3 = v_4$ (new)
State 1.4		$v_2 = v_5$ (new) <u>UNSATISFIABLE</u>	$v_2 = v_5$ (new)

Figure 4.1: Simulation of the Nelson and Oppen combination framework on the Formula 4.2 by propagation of equalities.

the new equality received that should also be considered now. After checking the satisfiability, the status still does not change and remains satisfiable, but a new equality is detected by \mathcal{T}_{LRA} -solver ($(v_1 = x \wedge x \leq y \wedge y \leq v_1) \implies x = y$).

State 1.2 - Once more, two decision procedures have a new constraint and can check for satisfiability. The status remains satisfiable, and a new equality is detected by \mathcal{T}_{UF} -solver ($(x = y \wedge f(x) = v_3 \wedge f(y) = v_4) \implies v_3 = v_4$).

State 1.3 - Again, new constraints are received and the status does not change. The new equality $v_2 = v_5$ is detected by \mathcal{T}_{LRA} -solver ($(v_3 = v_4 \wedge v_2 = v_3 - v_4 \implies v_2 = 0)$ and $(v_2 = 0 \wedge v_5 = 0 \implies v_2 = v_5)$).

State 1.4 - Finally, with the new equality $v_2 = v_5$, \mathcal{T}_{UF} -solver is able to detect the unsatisfiability due to a contradiction in $v_2 = v_5 \wedge P(v_2) \wedge \neg P(v_5)$. Formula 4.1 is therefore unsatisfiable.

As we have seen, this process is simple. Additionally, it is sound and complete. A proof is not simple but can be found in [68]. To make this procedure work, the main requirement is making the decision procedures deduce equalities between shared variables. However, for non-convex theories it gets more complicated as we will see in the following example.

4.1.2 Generation of disjunction of equalities and propagation

Consider the Formula 4.3. It is a mix of the convex theory of uninterpreted predicates (\mathcal{T}_{UF}) and the non-convex theory of linear arithmetic for integers (\mathcal{T}_{LIA}).

$$\phi : x \geq 0 \wedge x \leq 1 \wedge v_1 = 0 \wedge v_2 = 1 \wedge P(x) \wedge \neg P(v_1) \wedge \neg P(v_2) \quad (4.3)$$

In Figure 4.2 we can see a trace of a simulation to check for the satisfiability. We initially try to simulate Nelson and Oppen as before. At State 1.0, constraints are given to their respective decision procedures. No unsatisfiability is detected and no equalities can be deduced. However, the process should not stop since the problem is unsatisfiable. Clearly, the previous procedure is not complete in this case.

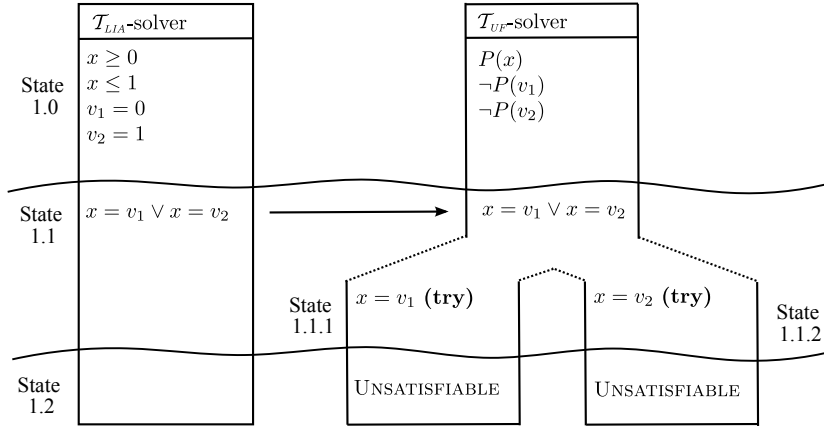


Figure 4.2: Simulation of the Nelson and Oppen combination framework on the Formula 4.3 by propagating disjunction of equalities.

The new fact is that from non-convex theories, even though no equalities can be deduced, disjunctions of equalities can still be. In this example, the disjunction $x = v_1 \vee x = v_2$ is a true deducible constraint, although neither of the single equalities $x = v_1$ nor $x = v_2$ is.

$x = v_1 \vee x = v_2$ is an important information and it is missing in the \mathcal{T}_{UF} -solver. At State 1.1, \mathcal{T}_{LIA} -solver generates and propagates this information to \mathcal{T}_{UF} -solver. Since \mathcal{T}_{UF} -solver cannot handle disjunctions in a direct way, it does by performing a case-split, where one of the equalities in the disjunction is tried at a time. If continuing generation and propagation of disjunction of equalities, there is a path that no contradiction can be found, then the problem is satisfiable. Otherwise, if all path lead to a contradiction, the problem is unsatisfiable. In this example, trying both $x = v_1$ and $x = v_2$ leads to a contradiction right after⁵, so the problem is unsatisfiable.

⁵ $x = v_1 \wedge P(x) \wedge \neg P(v_1) \implies \perp$; $x = v_2 \wedge P(x) \wedge \neg P(v_2) \implies \perp$.

In practice, SMT-solvers do not generate and propagate disjunctions of equalities. It is a complicated task to generate all disjunctions that would cover the missing information in the other decision procedures. Moreover, it is complicated and not efficient to handle disjunctions inside decision procedures.

4.1.3 Trying all arrangements

Exchanging disjunction of equalities is a theoretical alternative for arrangements. An arrangement in the Nelson and Oppen context, is a not immediately contradictory set that contains for each pair of variables x and y , either an equality $x = y$ or a disequality $x \neq y$. A brute-force, but complete way of combining disjoint and stably infinite theories is trying all the arrangements⁶ with the shared variables. If one of them is satisfiable, then the formula is also satisfiable. Otherwise, if none is satisfiable, the formula is unsatisfiable.

In the previous example (Formula 4.3), there are three shared variables: x , v_1 and v_2 . With them, five different arrangements⁷ can be built:

$$\begin{aligned}
 \mathcal{A}_1 &= \{x = v_1, x = v_2, v_1 = v_2\} \\
 \mathcal{A}_2 &= \{x = v_1, x \neq v_2, v_1 \neq v_2\} \\
 \mathcal{A}_3 &= \{x \neq v_1, x = v_2, v_1 \neq v_2\} \\
 \mathcal{A}_4 &= \{x \neq v_1, x \neq v_2, v_1 = v_2\} \\
 \mathcal{A}_5 &= \{x \neq v_1, x \neq v_2, v_1 \neq v_2\}
 \end{aligned} \tag{4.4}$$

In the beginning of the section, it was stated that for checking the satisfiability of two combined theories $\mathcal{T}_1 \cup \mathcal{T}_2$, we need to find a \mathcal{L} such that the problem could be reduced to checking the satisfiability of $\mathcal{T}_1 \cup \mathcal{L}$ and $\mathcal{T}_2 \cup \mathcal{L}$, separately. Determining \mathcal{L} is easy for convex theories, but it is unpractical for non-convex theories. The idea of the arrangements is that if we cannot determine \mathcal{L} , trying all the arrangements will solve the case. So, in the case

⁶That is the number of ways a set of n elements can be partitioned into nonempty subsets, also know in combinatorics as Bell numbers. This numbers grow very fast. For the first few n , the Bell numbers are 1, 2, 5, 15, 52, 203, 877, 4140, 21147, 115975, ...

They can be generated using the recursive formula $B_n = \sum_{k=0}^{n-1} B_k \binom{n-1}{k}$.

⁷A set that is contradictory by itself, like $\{x = v_1, x = v_2, v_1 \neq v_2\}$, is not considered an arrangement.

of the last example, \mathcal{L} is:

$$\begin{aligned}
\mathcal{L} = & (x = v_1 \wedge x = v_2 \wedge v_1 = v_2) \\
& \vee (x = v_1 \wedge x \neq v_2 \wedge v_1 \neq v_2) \\
& \vee (x \neq v_1 \wedge x = v_2 \wedge v_1 \neq v_2) \\
& \vee (x \neq v_1 \wedge x \neq v_2 \wedge v_1 = v_2) \\
& \vee (x \neq v_1 \wedge x \neq v_2 \wedge v_1 \neq v_2)
\end{aligned} \tag{4.5}$$

The satisfiability can be checked using only the decision procedures by separating the theories in the formula, and performing a check with these five arrangements in 4.4. However, using an SMT-solver is preferable because one can make use of important techniques such as learning, backtracking, etc. that can help reducing the search space and efficiently perform case splits. For making use of an SMT-solver, \mathcal{L} will be converted to CNF. The result⁸ of the conversion can be seen in the Formula 4.6.

$$\begin{aligned}
\mathcal{L}' = & (x = v_1 \vee x \neq v_2 \vee v_1 \neq v_2) \\
& \wedge (x \neq v_1 \vee x = v_2 \vee v_1 \neq v_2) \\
& \wedge (x \neq v_1 \vee x \neq v_2 \vee v_1 = v_2)
\end{aligned} \tag{4.6}$$

Reformulating Formula 4.3, we get Formula 4.7. Now, an SMT-solver with no cooperation between the decision procedures can check for the formula satisfiability.

$$\phi' : \phi \wedge \mathcal{L}' \tag{4.7}$$

The continuation of the example, now using arrangements, is shown in Figure 4.3. At each state, the SAT-solver sends a propositionally satisfiable assignment⁹ to the decision procedures, that will handle the constraints relevant to their theory. At each iteration one of the decision procedures finds a conflict and sends it back to the SAT-solver, that learns the new clause. By learning new clauses, at State 5, the SAT-solver cannot find any more propositionally satisfiable assignments and concludes that the problem is unsatisfiable.

In principle, each assignment should correspond to one arrangement. But notice that only four assignments were tried, instead of the five expected. That is because learning the new clauses allowed the SAT-solver to reduce the search space. In this case, after $(\neg P(x) \vee P(v_1) \vee x \neq v_1)$ was learned at State 2, the assignments containing $x = v_1$ were no longer propositionally satisfiable, and therefore, the assignment that would contain the arrangement \mathcal{A}_2 was never sent to the decision procedures.

⁸ \mathcal{L}' can be obtained by applying distributivity, resulting in a very large formula, and later removing redundant expressions.

⁹There is a propositionally satisfiable assignment when, ignoring the theory of the literals, we can assign each of the literals to either *true* or *false* in such a way as to make the formula evaluate to *true*.

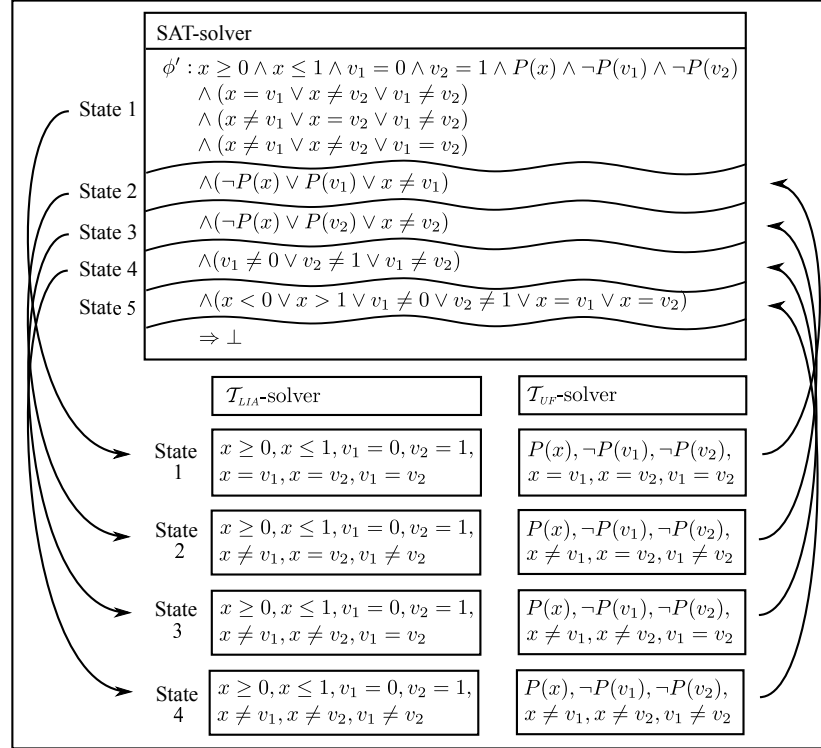


Figure 4.3: Simulation of the Nelson and Oppen combination framework on the Formula 4.7 by using arrangements. The SAT-solver plays an important role, learning clauses and performing the case splits.

4.1.4 Remarks

Using arrangements like presented here, solvers are complete for combinations of disjoint and stably infinite theories. That is the basis of techniques like *delayed theory propagation*, see e.g. [16], that many SMT-solvers use. One may get the impression that a lot of effort is necessary when the number of shared variables is high, but the use of the SAT-solver with learning techniques helps considerably the process [17].

Testing all arrangements is not necessary when theories are convex because the arrangement is implicitly deduced in this case. Part of the set is the equalities that were generated by the decision procedures. The remaining is simply the disequalities between every pair of variables that were not stated as equal by any of the decision procedures. The same reasoning cannot be done for a non-convex theory. If for instance, neither of the single equalities $x = v_1$ nor $x = v_2$ were deduced, we cannot say that $x \neq v_1$ and $x \neq v_2$ when, from the theory, we could have extracted the disjunction $x = v_1 \vee x = v_2$.

One final practical observation of Nelson and Oppen is its good modularity. We can easily incorporate new decision procedures and increase the expressiveness of the solver. Also, replace a decision procedure by a new one.

In the next section we introduce the notion of *model-equalities*, that works like an extension to the Nelson and Oppen equality propagation method. It is an alternative for the combination of non-convex theories, that avoids enumerating all arrangements and guides the satisfiability search by using models inside the theories.

4.2 Combination sharing model-equalities

In the previous section, we saw some ways of combining theories. They work well in some cases and had limitations in others. In this section, we present an original alternative for combining disjoint and stably infinite theories, that is well suited for non-convex theories, *combining decision procedures by model-equalities propagation* [26].

The aim of combining decision procedures by *model-equalities* is to find an arrangement of the shared variables, being guided by the decision procedures instead of blindly guessing one, like it was presented in the previous section. The idea is that the decision procedures will keep a model, or build a consistent one when required, and look at the values of the variables in this model to generate equalities between variables. We call these equalities, created from models, *model-equalities*. They are propagated to other decision procedures that will incorporate them and continue the process.

First let us focus on understanding the models. To illustrate the idea, take a look at the next example. The linear arithmetic decision procedure over integers contains the following constraints:

$$x \geq 0, x \leq 1, v_1 = 0, v_2 = 1 \quad (4.8)$$

This example is the same as Example 4.3 in previous section. We know that no equality can be deduced from it. There are only two models that are consistent in this case. They can be seen in Table 4.1. The way a model is generated depends on the decision procedure. The details of how to build a model for difference logic and linear arithmetic theories will be given in Chapters 6 and 7.

From Model 1, both x and v_1 have the same value. Therefore, we can extract the model-equality $x = v_1$. In the case of the Model 2, $x = v_2$ can be extracted.

The simulation (example of Equation 4.3) of an SMT-solver using propagation of model-equalities can be seen in Figure 4.4. One can notice that, comparing to the simulation of Figure 4.3, the non extended version of Formula 4.3 is been used.

Model 1		Model 2	
Variable	Value	Variable	Value
x	0	x	1
v_1	0	v_1	0
v_2	1	v_2	1

Table 4.1: Arithmetic models from Example 4.8.

There are many similarities in both simulations. For instance, in both of them, the SAT-solver learns during the process. At State 2 and 3, it learns that x should be different from v_1 and v_2 .

One difference between the simulations is that in the one of Figure 4.3, there is an equality or disequality between each pair of shared variables in each assignment given by the SAT-solver. While in Figure 4.4, this arrangement is being built during the learning process.

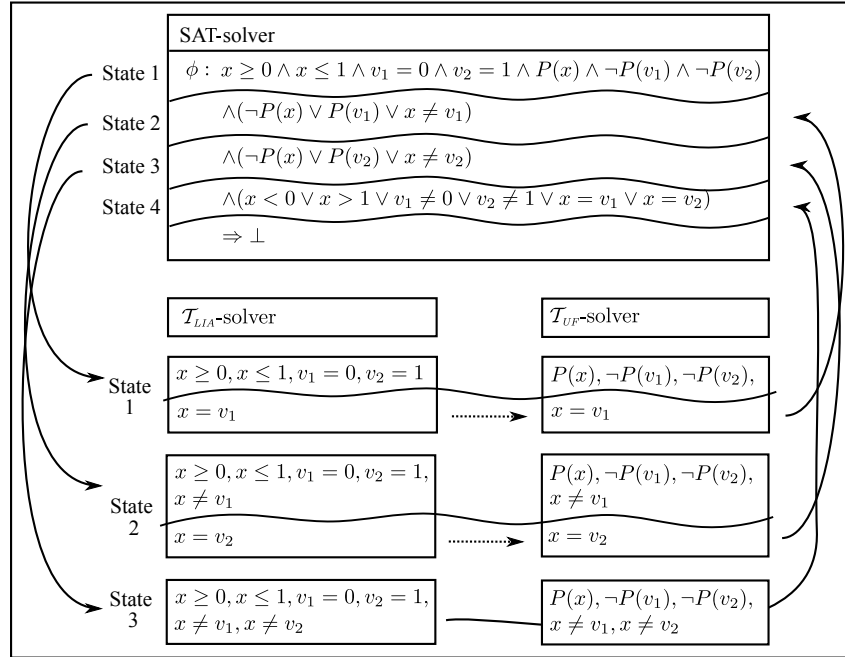


Figure 4.4: Simulation of the combination by model-equality propagation on the Formula 4.3. The \mathcal{T}_{LIA} -solver generates model-equalities and propagates to the \mathcal{T}_{UF} -solver. Learning conflicts, the SAT-solver finds the unsatisfiability at State 4.

An advantage of the second simulation is that, using the decision procedures to build this missing information, the SAT-solver can avoid to guess

theory inconsistent literals, like $v_1 = v_2$ ¹⁰. This allows the SMT-solver to potentially reduce the number of assignments tried.

Another important observation is that the non-convexity is implicitly handled by the models. Take a look at this example, the disjunction $x = v_1 \vee x = v_2$ is hidden inside the arithmetic constraints. However, for every arithmetic model, one of such equalities will be propagated, i.e., there is no model where $x \neq v_1 \wedge x \neq v_2$ is true.

In practice, the combination using model-equalities is very close to the combination using only equalities. It is relatively easy to adapt an SMT-solver to work using model-equalities. Abstracting ourselves from the details of the decision procedures (like how to generate equalities and model-equalities) that are going to be presented in future chapters, the general procedure can be described like this:

- The SAT-solver finds a propositionally satisfiable assignment and sends it to the decision procedures.
- Each decision procedure gets its relevant constraints and check for unsatisfiability.
- If unsatisfiability is not found by any of the decision procedures, they repeatedly look for equalities and propagate them to the other decision procedures.
- Meanwhile, if no more equalities can be deduced and unsatisfiability was not found, the decision procedures of non-convex theories generate model-equalities and propagate them.
- If after all equalities and model-equalities have been propagated, no conflict arises, the problem is satisfiable.
- Whenever a conflict is found, a conflict clause is generated, added to the working formula in the SAT-solver, and a new assignment is produced.
- If the SAT-solver cannot find a propositionally satisfiable assignment, the problem is unsatisfiable.

In addition to working with model-equalities, a practical change in this combination is that, since model-equalities are very often new¹¹ literals, conflict sets including model-equalities may introduce new literals in the SAT-solver. However, this does not affect termination, since the number of literals that may be created are limited by the number of variables in the input formula.

¹⁰ \mathcal{T}_{LIA} -solver knowing that $v_1 = 0$ and $v_2 = 1$, will never generate $v_1 = v_2$.

¹¹Not present in the original formula

An example of combining the theory of uninterpreted functions and integer difference logic

In this section, we present an example illustrating the cooperation using *model-equalities* between the theory of uninterpreted functions (UF) and the fragment of linear arithmetic, integer difference logic (IDL). The simulation done is very close to what really happens inside our SMT-solver `veriT` [14]. Difference Logic is the linear arithmetic fragment that contains only constraints of the kind $x - y \bowtie c$, where x and y are variables, c is a constant number and $\bowtie \in \{\leq, \geq, =, <, >\}$. Difference logic and decision procedures for difference logic will be explained in details in Chapters 6.1 and 6.2. Here, we look at them in a superficial way to not shift the focus of the chapter. Assume we want to prove that the following formula is unsatisfiable:

$$x \leq y + 1 \wedge y \leq x \wedge x \neq y \wedge f(x) \neq f(y + 1) \quad (4.9)$$

As a first step and for simplicity of the presentation, the formula is purified (i.e. the *separation* is done at the formula level) so that the different decision procedures only get literals with symbols from their theory.¹²

$$\underbrace{v_1 = y + 1}_{p_1} \wedge \underbrace{x \leq v_1}_{p_2} \wedge \underbrace{y \leq x}_{p_3} \wedge \underbrace{x \neq y}_{\neg p_4} \wedge \underbrace{f(x) \neq f(v_1)}_{\neg p_5} \quad (4.10)$$

Every atom is assigned a propositional variable p_i .

Figure 4.5 can be used to trace the status of the algorithm during its application to this problem. In Figure 4.5 and in the following, the symbols p_i may be used to denote the constraints to which they correspond. Also $a \neq b$ denotes $\neg(a = b)$, and $a > b$ (or $a < b$) may be used instead of $\neg(a \leq b)$ (respectively $\neg(a \geq b)$). \mathcal{T}_{UF} and \mathcal{T}_{IDL} are the theories for uninterpreted functions (UF) and integer difference logic (IDL) respectively.

State 1.0 The SAT-solver has found an entailing assignment for the formula.

In this assignment, unsurprisingly, p_1 , p_2 and p_3 are assigned to true, whereas p_4 and p_5 are assigned to false. This assignment is propagated to the decision procedures to check for theory consistency. At this point, no equality can be generated. The process would stop here if there were only convex theories involved.

State 1.1 However, IDL is non-convex. To obtain completeness of the cooperation of the decision procedures, one can generate model-equalities¹³.

A suitable model for State 1.0 assigns $x = 0$, $y = -1$ and $v_1 = 0$. The

¹²Another approach, used in `veriT`, is to build the separation on-the-fly at the theory reasoner level.

¹³Details of how this can be done in a difference logic decision procedure are in Section 6.2.5.

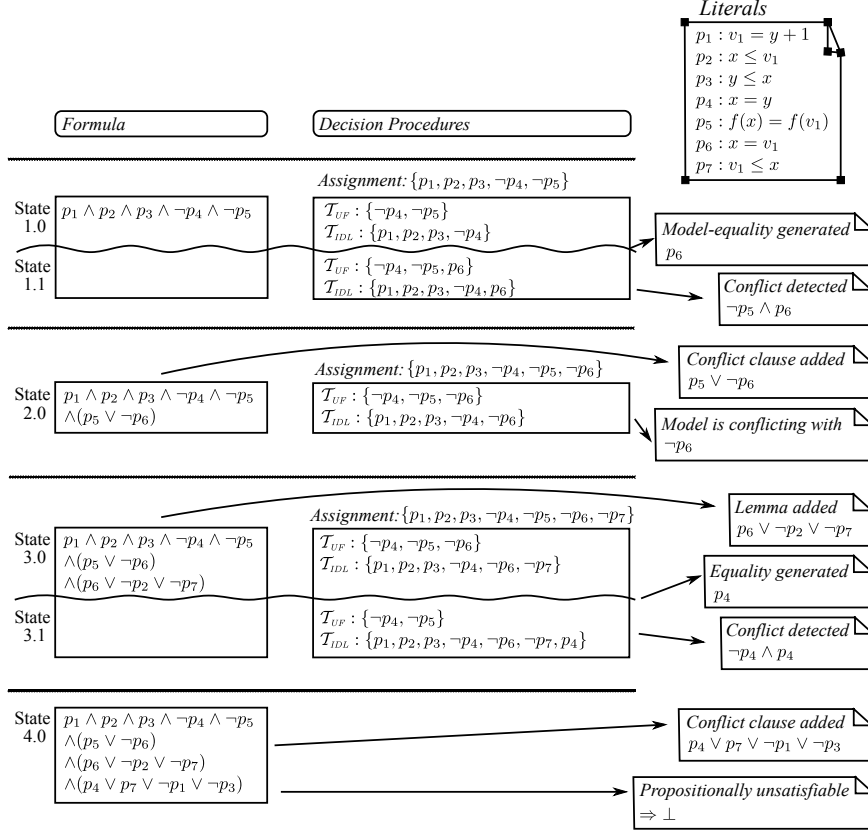


Figure 4.5: An example combining UF and IDL.

model equality $x = v_1$ (abstracted to a new proposition p_6) is generated and propagated to all decision procedures. The resulting state is State 1.1, and a contradiction is found in the \mathcal{T}_{UF} -solver, since the conjunction $x = v_1 \wedge f(x) \neq f(v_1)$ is unsatisfiable. The conflict clause $p_5 \vee \neg p_6$ is added to the SAT-solver.

State 2.0 A second iteration is necessary. The SAT-solver is asked for a new propositional assignment, the decision procedures backtrack to the previous satisfiable state, i.e. just before the model equality $x = v_1$ (i.e. p_6) was propagated. The set of literals for each decision procedure is updated to reflect the change in the SAT-solver assignment: the literal $\neg p_6$ (i.e. $x \neq v_1$) is added to both sets. No contradiction or equality is generated by either decision procedure, but the current assignment of concrete values to variables for IDL is not consistent with the current assignment from the SAT solver, in particular since it contradicts $\neg p_6$ (i.e. $x \neq v_1$). Our implementation of the IDL decision procedure is not able to automatically handle negation of equalities; in order

to repair the model, the IDL decision procedure generates a *lemma*: $(x = v_1) \vee (x > v_1) \vee (x < v_1)$ (or equivalently $p_6 \vee \neg p_2 \vee \neg p_7$, where p_7 is a new propositional variable corresponding to the atom $v_1 \leq x$).

State 3.0 The lemma is given to the SAT-solver, which therefore refines the propositional assignment to include $\neg p_7$. At this point, the IDL decision procedure is able to deduce the *equality between shared variables* $x = y$ from $x < v_1$, $v_1 = y + 1$ and $y \leq x$.

State 3.1 The equality $x = y$ is propagated to the \mathcal{T}_{UF} -solver, which detects the conflict $x \neq y \wedge x = y$, so the assignment is once again considered theory inconsistent. The corresponding conflict clause¹⁴ is generated ($p_4 \vee p_7 \vee \neg p_1 \vee \neg p_3$) and added to the SAT-solver.

State 4.0 Finally, the SAT-solver concludes there is no more assignment to make the current formula propositionally true. Therefore, the problem is unsatisfiable.

An example of combining uninterpreted functions with non-linear arithmetic

Any non-convex theory can make use of model-equalities to achieve completeness when cooperating with other theories. The previous example highlights the cooperation between the SAT-solver and the combination of uninterpreted functions and integer difference logic theories. In this section, we show that the theory of uninterpreted functions (UF) and non-linear arithmetic (NLA) can cooperate by exchanging equalities and model-equalities. Our SMT-solver, veriT, does not contain a decision procedure for non-linear arithmetic. However, with this example, we want to show that it should not be hard to expand the use of model-equalities to other decision procedures.

The details of the interplay between the theory reasoners and the SAT-solver are abstracted to focus on the equality exchanges between the decision procedures. The internal details of the decision procedures are also quietly ignored. It is just assumed that it is possible to retrieve implied equalities from the constraint set and to detect unsatisfiability. It is also assumed that the non-linear arithmetic decision procedure can maintain a concrete model, assigning values to variables. This model will be helpful when generating model-equalities.

To study the satisfiability of the following formula

$$x^2 = 1 \wedge y^2 = 4 \wedge f(2x) = 1 \wedge f(y) = 0 \wedge f(-y) = 0, \quad (4.11)$$

¹⁴The deduced equality $x = y$ is not used directly in the conflict clause. The literals that originated it, $\neg p_7 \wedge p_1 \wedge p_3$ are used instead.

like in the previous example, the formula is first purified:

$$x^2 = 1 \wedge y^2 = 4 \wedge v_1 = 2x \wedge v_2 = -y \wedge v_3 = 1 \wedge v_4 = 0 \wedge f(v_2) = v_4 \wedge f(v_1) = v_3 \wedge f(y) = v_4 \quad (4.12)$$

The constraints are first dispatched to their respective decision procedures, \mathcal{T}_{NLA} -solver for non-linear arithmetic, and \mathcal{T}_{UF} -solver for uninterpreted symbols. This is shown in Figure 4.6, at State 1.0.

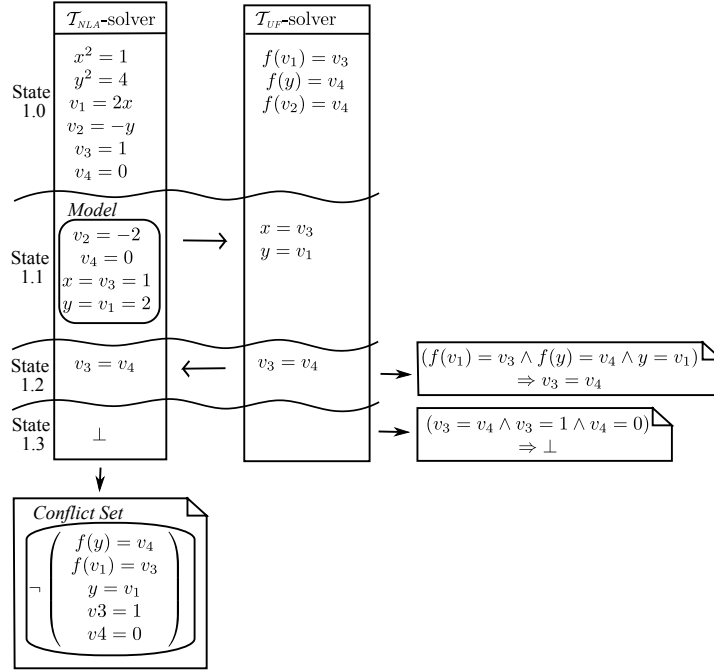


Figure 4.6: Example UF and NLA: Trying the first assignment.

At State 1.0, both decision procedures for \mathcal{T}_{NLA} and \mathcal{T}_{UF} conclude that their set of constraints are satisfiable. Furthermore, no equality between variables can be deduced.

However \mathcal{T}_{NLA} is not convex. It is not correct to conclude, in State 1.0, that the set of constraints is satisfiable. At State 1.1, the decision procedure for \mathcal{T}_{NLA} has generated the arithmetic model $v_2 = -2$, $v_4 = 0$, $x = v_3 = 1$ and $y = v_1 = 2$, and thus can indeed generate two model-equalities that are propagated to \mathcal{T}_{UF} -solver. At State 1.2, \mathcal{T}_{UF} deduces the equality $v_3 = v_4$ that results in a conflict on the arithmetic side, once propagated. This conflict is translated to a clause and sent to the SAT-solver which will learn about the theory inconsistency and will generate a new assignment.

This new assignment will naturally assign false to the new constraint $y = v_1$, in order to make the new conflict clause true (Figure 4.7, state 2.0). The iteration process is similar, but because of the new constraint,

the model generated by the decision procedure for \mathcal{T}_{NLA} is different from the previous one, which results in yet another model-equality. The final result is the same: after the deductions and propagations, \mathcal{T}_{NLA} -solver finds a conflict.

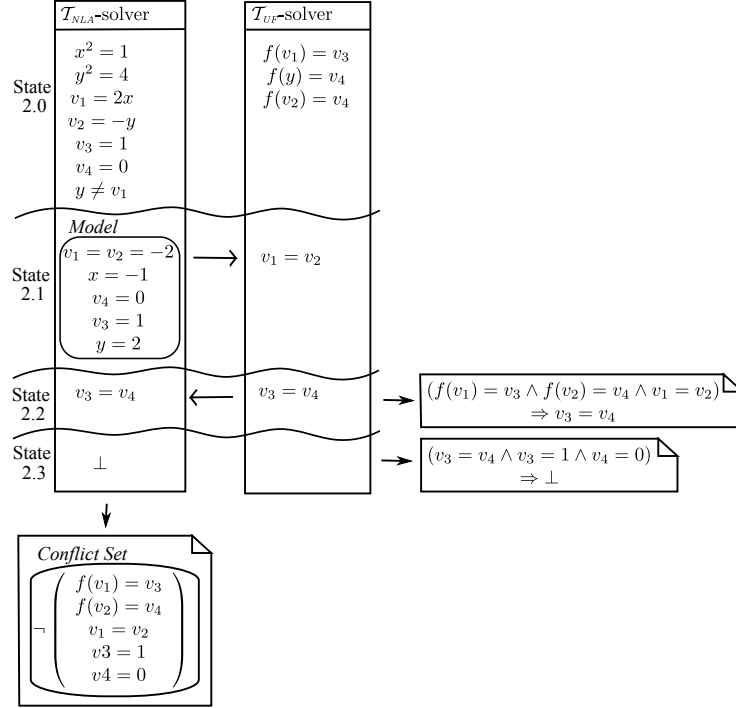


Figure 4.7: Example UF and NLA: Trying the second assignment.

The third assignment (Figure 4.8, State 3.0) differs from the previous one by one new constraint only. Now \mathcal{T}_{NLA} -solver finds an inconsistency directly, no arithmetic model can be found anymore. Including this new conflict clause will result in a formula which is propositionally unsatisfiable. Therefore the original formula is unsatisfiable.

It is worth noticing that, by combination of all possible values of x and y (according to constraints $x^2 = 1$ and $y^2 = 4$), four different arithmetic models are possible. But thanks to the arithmetic reasoning and learning process, just two of them need to be examined to conclude the unsatisfiability of the formula.

4.3 Combining with model-equalities, an algorithm

In Algorithm 3, we propose a high level pseudo-code of the Nelson and Oppen framework with model-equalities. It reflects pretty much how the previous examples worked and has minimal changes if compared to the original framework.

4.3. COMBINING WITH MODEL-EQUALITIES, AN ALGORITHM 75

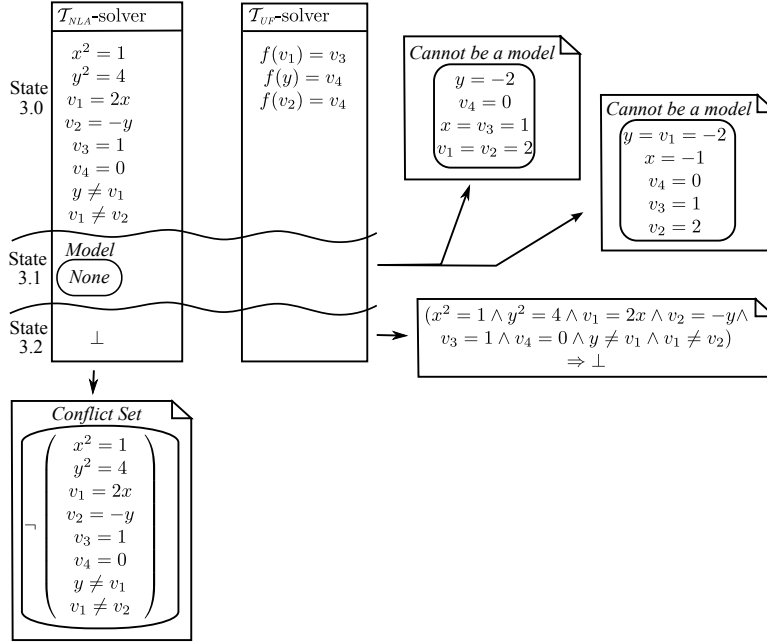


Figure 4.8: Example UF and NLA: Trying the third and last assignment.

The propositional satisfiability solver should be able to incorporate new literals on-the-fly (corresponding to constraints that are not in the original formula). Also, as two consecutive assignments may have many common literals, the presented algorithm gives the decision procedures the opportunity to take advantage of the similarities between consecutive sets of literals produced by the propositional SAT-solver as they may update their state to reflect only the difference between these sets. This is the case when there exists efficient decision procedures that incrementally stack new literals and backtrack while maintaining the \mathcal{T} -satisfiability status of the current set of literals.

The set of theories in the combination is denoted $S_{\mathcal{T}}$, and the subset of theories that are not convex or with a decision procedure that is not able to deduce all the implied equalities is denoted $S_{\mathcal{T},\text{mod}}$. It is assumed that the decision procedure for each theory in $S_{\mathcal{T},\text{mod}}$ is able to generate the model-equalities from a model they maintain based on the literals (including equalities and inequalities) they receive. The main difference of Algorithm 3 with respect to a version without model equalities is located in the lines 16 to 18.

The main loop of the algorithm is executed until the SAT-solver can no longer produce a propositional satisfiable assignment (line 1). In this case, the original formula is unsatisfiable. Otherwise, a propositional model is computed (line 2). Each decision procedure t may then backtrack to

```

output: status: {(SAT, assignment); UNSAT }
data : assignment: Assignment
data : lemmas: Set of Lemma
data : newLiterals: Set of Literal
data : SAT_solver: SATSolver

1 while SAT_solver.GetStatus() = SAT do
2   assignment := SAT_solver.assignment();
3   status := SAT ;
4   foreach t in  $S_{\mathcal{T}}$  do t.Backjump(assignment);
5   newLiterals := assignment.GetNewLiterals();
6   repeat
7     foreach t in  $S_{\mathcal{T}}$  do
8       status := t.Propagate(newLiterals);
9       if status = UNSAT then
10        SAT_solver.Add(t.conflictClause);
11        break ;
12      end
13    end
14    if status = UNSAT then break ;
15    newLiterals :=  $\bigcup_{t \in S_{\mathcal{T}}} t.GetNewEqualities()$ ;
16    if newLiterals =  $\emptyset$  then
17      newLiterals :=  $\bigcup_{t \in S_{\mathcal{T}, \text{mod}}} t.GetNewModelEqualities()$ ;
18    end
19  until newLiterals =  $\emptyset$  ;
20  lemmas :=  $\bigcup_{t \in S_{\mathcal{T}}} t.GetNewLemmas()$ ;
21  SAT_solver.Add(lemmas);
22  if lemmas =  $\emptyset$   $\wedge$  status = SAT  $\wedge$  assignment.IsTotal() then
23    return (SAT, assignment);
24  end
25 end
26 return UNSAT ;

```

Algorithm 3: A SMT-solver using Nelson and Oppen combination framework plus model-equality generation and propagation.

a state based on the new set of literals corresponding to this assignment (line 4). Note that the set of literals available in such state should be \mathcal{T}_i -satisfiable in each theory \mathcal{T}_i . The variable *newLiterals* represents the set of literals that the decision procedures need to receive. It is initialized with the new literals present in the assignment (line 5), and is later updated with equalities produced by the decision procedures (lines 15 and 17). This set is repeatedly propagated to each decision procedure through the *Propagate()* function (line 8) until one of them detects unsatisfiability (line 9) or no new equalities can be deduced (line 19). If unsatisfiability is detected, then a conflict clause is generated and added to the propositional satisfiability solver (line 10). Otherwise, each decision procedure computes the set of

variable equalities entailed by the current set of literals. These sets are stored (line 15) for propagation at the next iteration. Ultimately, if no equalities between variables can be deduced, then the decision procedures of non-convex¹⁵ theories will look for model-equalities to propagate.

Once all the literals and equalities have been propagated, additional lemmas produced by the decision procedures may be incorporated as clauses to the propositional satisfiability solver (lines 20, 21). Eventually, when no new information can be provided to the propositional satisfiability solver, and if the assignment is total, then the algorithm concludes that the original formula is \mathcal{T} -satisfiable and halts (line 23).

4.4 Soundness and completeness of SMT-solvers

In this section, we show a proof for soundness and completeness of SMT-solvers and model-equalities like presented in [26]. Initially, the formula given as input to the SMT-solver is converted to a conjunctive set of clauses S . The goal of the SMT-solver is to decide whether S is \mathcal{T} -unsatisfiable or that S satisfiable. In the former case, the SMT-solver is said to be in the final state UNSAT. In the later case, the SMT-solver shall additionally identify a boolean assignment Γ of the atoms occurring in S , such that S is \mathcal{T} -satisfiable. This corresponds to the final state denoted SAT(Γ). A sound SMT-solver will never get to the UNSAT final state on a \mathcal{T} -satisfiable formula, or decide satisfiability of a \mathcal{T} -unsatisfiable formula. It is complete if it always terminates.

The SAT-solver maintains a boolean assignment Γ of the atoms in the set of clauses. The pair $\langle S, \Gamma \rangle$ thus represents the current intermediate state of the solver. The set of rules given in Figure 4.9, given in a SOS-like style, provides an abstract non-deterministic model of the possible behavior of the SMT-solver and is described in details in the following. Observe that clauses may be added to the set S either by the SAT-solver itself, or by the theory reasoner (based on the propositional assignment from the SAT-solver). The reasoning ends when the SAT-solver concludes that the set of clauses is unsatisfiable, or when the theory reasoner asserts that the propositional assignment Γ is also \mathcal{T} -satisfiable.

Rule BOOL (4.13) formalizes the update of the propositional assignment by the SAT-solver; Γ' is a new assignment such that $\Gamma' \cup \{C\}$ is propositionally satisfiable for every clause $C \in S$. The assignment is not required to be total; an assumption about assignment totality will be made later. The SAT-solver can also conclude that S is unsatisfiable using rule UNSAT (4.14).

The addition of new clauses is represented by rule LEARN (4.15). The

¹⁵In practice, decision procedures that do not have capacity of detecting all implied equalities may also use model-equalities to achieve completeness, since the models will naturally include the equalities.

$$\text{BOOL: } \frac{\langle S, \Gamma \rangle}{\langle S, \Gamma' \rangle} \quad \Gamma' \text{ is a propositional assignment of } S \quad (4.13)$$

$$\text{UNSAT: } \frac{\langle S, \Gamma \rangle}{\text{UNSAT}} \quad S \text{ is propositionally unsatisfiable} \quad (4.14)$$

$$\text{LEARN: } \frac{\langle S, \Gamma \rangle}{\langle S \cup \{C\}, \Gamma \rangle} \quad S \models_{\mathcal{T}} C \quad (4.15)$$

$$\text{SAT: } \frac{\langle S, \Gamma \rangle}{\text{SAT}(\Gamma)} \quad \Gamma \text{ is entailing and } \mathcal{T}\text{-satisfiable} \quad (4.16)$$

Figure 4.9: Rules representing the execution of an SMT-solver

new clause C may be added by the SAT-solver itself. In that case, C is a propositional consequence of S . The clause may also be added by the theory reasoner; C should then be a \mathcal{T} -logical consequence of the set S , according to the considered theory ($S \models_{\mathcal{T}} C$). By induction, it is clear that every added clause is a consequence of the original formula, and that the set of clauses is always \mathcal{T} -equisatisfiable to the original set of clauses.

When the assignment produced by the SAT-solver is entailing and \mathcal{T} -satisfiable, then the theory solver may conclude that the formula is \mathcal{T} -satisfiable. This is summarized in rule SAT (4.16).

In the present scheme, no assumption is made on the order of application of rules, on how the clause C is generated in LEARN (4.15) and on the relation between consecutive assignments from the SAT-solver. This is all left abstract, with side conditions for the soundness and completeness of the SMT-solver.

Theorem 1. *An SMT-solver implementing the rules of Figure 4.9 is sound.*

Proof. Initially, the set of clauses is just a conjunctive normal form of the input formula. We first prove by induction that the set of clauses given to the SAT-solver is always \mathcal{T} -equisatisfiable to the input formula. Assume that S and S' are the sets of clauses respectively before and after the application of a rule of Figure 4.9. The sets S and S' only differ when rule LEARN (4.15) is applied. In that case $S' = S \cup \{C\}$, with $S \models_{\mathcal{T}} C$. Thus S' is a \mathcal{T} -logical consequence of S ; conversely, since S' contains S , S is also a \mathcal{T} -logical consequence of S' . In other words S and S' are \mathcal{T} -logically equivalent. By induction, the set of clauses is always \mathcal{T} -logically equivalent to the original set of clauses, and thus \mathcal{T} -equisatisfiable to the input formula.

If the SAT-solver concludes that the set of clauses is propositionally unsatisfiable (using rule UNSAT (4.14)), the initial set of clauses and the input formula are unsatisfiable.

If rule SAT (4.16) is applied, then there exists a \mathcal{T} -satisfiable entailing assignment Γ of the set of clauses S . Assume \mathcal{M} is a \mathcal{T} -model of Γ . Since Γ is a propositional model of S , \mathcal{M} is a model of every clause in S . The set of clauses S , like the original formula, is thus satisfiable. \square

Notice that the assumption in rule LEARN (4.15) is very permissive. It holds notably for propositional learning, a technique used inside SAT-solvers, where the new clause C is obtained by propositional resolution of clauses in S , guided by a conflict analysis procedure known as the FUIP (First Unique Implication Point) computation [73]. It also holds for conflict clauses from the theory reasoner where the clause C is the disjunction of the negation of literals in a \mathcal{T} -unsatisfiable subset of the assignment Γ (\mathcal{T} being the considered theory). Some further assumptions are however required to prove the completeness of an SMT-solver implementing the rules of Figure 4.9.

Theorem 2. *An SMT-solver implementing the rules of Figure 4.9 is complete (eventually terminates on a SAT(Γ) or UNSAT state) provided that*

- *on any set of clauses, the SAT-solver will eventually either*
 - *provide an entailing assignment*
 - *or conclude the unsatisfiability of the set of clauses with rule UNSAT (4.14);*
- *the atoms of the clauses added by rule LEARN (4.15) belong to a finite set that is fixed a priori for the whole execution of the SMT-solver;*
- *for any state $\langle S, \Gamma \rangle$ where Γ is entailing, either rule SAT (4.16) is applied or rule LEARN (4.15) is applied, with C not being a propositional consequence of Γ .*

Proof. If the run is finite then it should end either with rule SAT (4.16) or rule UNSAT (4.14). This is proved by contradiction. Assume the run is finite but does not terminate on an UNSAT or SAT state. Then the ending state is of the form $\langle S, \Gamma \rangle$. The first assumption implies that Γ is entailing. Since Γ is entailing, the last assumption ensures either that

- the new state is SAT (with the application of rule SAT (4.16)) or
- the rule LEARN (4.15) is applied and introduces a clause C that is not a propositional consequence of Γ .

The first option is not possible, since it contradicts the hypothesis that the ending state is not a SAT state. The second option is also not possible, since this would change the set S , and contradicts the fact that $\langle S, \Gamma \rangle$ is the ending state.

Assume now that the run is infinite. The set of atoms that are or will be present in the set of clauses is finite, thanks to the second assumption. The

set of possible different clauses is also finite. At some point no new clause will be added to the set of clauses S by rule LEARN (4.15), and the SAT-solver will eventually provide an entailing assignment Γ . The rule SAT (4.16) being an ending rule, the next rule will be rule LEARN (4.15), and a clause C will be generated. Since C already belongs to the set of clauses and Γ is entailing, then C is a logical consequence of Γ which contradicts the last assumption of the theorem. \square

The three requirements in the above theorem are reasonable. The first requirement is on the SAT-solver: on any set of clauses, it should decide that it is unsatisfiable, or provide a total (and thus entailing) assignment. This requirement is fulfilled by existing tools [49, 32]. The two remaining requirements are related to the theory reasoner and are discussed in the next subsection.

4.4.1 Model-equalities

The model-equality method presented in this chapter is suitable for decision procedures that are not able to deduce disjunctions of equalities, or that are not complete with respect to deduction of (disjunctions of) equalities. We assume they are however able to find a concrete model for a set of constraints, i.e. literals. As an example, it means that a decision procedure for integer linear arithmetic is able to find a mapping from variables to integers such that all constraints are satisfiable. Many decision procedures inherently have such a capability.

Assume that an assignment Γ provided by the SAT-solver produces (pure) literals Γ_1 and Γ_2 to be handled by theory reasoners for \mathcal{T}_1 and \mathcal{T}_2 respectively. Assume also that Γ_1 is \mathcal{T}_1 -satisfiable, and Γ_2 is \mathcal{T}_2 -satisfiable. Finally assume that all generated disjunctions of equalities have been handled as in the previous subsection. The theory reasoners that are not complete with respect to deduction of (disjunctions of) equalities should then compute a model, and generate the equalities between shared variables that correspond to the model and that do not already belong to Γ . Those equalities are then given to the other decision procedure, as if they were in the original assignment. Those equalities may themselves force the other decision procedure to deduce or produce other equalities.

Theorem 3. *Consider two stably infinite and disjoint theories T_1 and T_2 . Assume that the decision procedures for each of those theories are complete, and are complete for the generation of model-equalities. An SMT-solver combining both decision procedures and verifying condition 2 of Theorem 2 is (sound and) complete.*

Proof. The result is a direct consequence of theorem 2. Indeed all hypotheses of 2 are fulfilled. The first hypothesis is explicitly an hypothesis of the present theorem.

For the second hypothesis, conflict clauses generated in an SMT-solver using model-equalities will contain terms from the original formula or model equalities. There is a finite number of possible model-equalities since they are equalities between terms in the original formula only. Hence the atoms in conflict clauses belong to a finite set that depends on the original formula only.

For the last condition of theorem 2, remember that the decision procedures are complete for generation of model-equalities. At each step, for each decision procedure, one of the following cases holds:

- unsatisfiability of the set of known literals can be deduced;
- an equality that is not in the known literals can be generated;
- a model-equality that is not in the known literals can be generated;
- or the set of literals is satisfiable and no more equality or model-equality can be deduced.

Assume Γ_1 and Γ_2 are the set of literals given respectively to the decision procedures for \mathcal{T}_1 and \mathcal{T}_2 from the purification of the set of literals Γ , and that all deduced equalities have been propagated, and model-equalities have been generated. If a conflict occurs, the theory reasoner for \mathcal{T}_i generates a conflict clause C of the form $\bigvee_{\ell \in \gamma} \neg \ell$ where γ is a \mathcal{T}_i -unsatisfiable subset of $\Gamma \cup \Gamma'$ with Γ' being the set of all generated equalities and model-equalities. This clause is added to the set of clauses handled by the SAT-solver. It may contain atoms (equalities) coming from models. If it does not, it is conflicting in the sense that $\Gamma \cup \{C\}$ is unsatisfiable. If it does, it is obviously not a propositional consequence of Γ .

If no conflict occurs, then $\Gamma \cup \Gamma'$ contains equalities between any two shared variables that are equal according to the model. Conversely, if an equality between two shared variables does not belong to $\Gamma \cup \Gamma'$, it has not been guessed nor deduced by the decision procedures in the combination. If we assume every decision procedure is either complete with respect to the generation of disjunction of equalities, or that it generates model equalities, one can conclude that the two theories agree that, if no equality between two shared variables exists in $\Gamma \cup \Gamma'$, they should be different. An arrangement \mathcal{A} can thus be built from the equalities in $\Gamma \cup \Gamma'$, augmented by the maximum number of disequalities between shared variables. $\mathcal{A} \cup \Gamma_1$ is \mathcal{T}_1 -satisfiable and $\mathcal{A} \cup \Gamma_2$ is \mathcal{T}_2 -satisfiable. Thanks to the combination based on the Nelson and Oppen framework [53] for stably infinite and disjoint theories, $\mathcal{A} \cup \Gamma$ is also satisfiable in the union of theories. Rule SAT (4.16) can be applied and the third assumption of Theorem 2 is fulfilled. The approach is (sound and) complete. □

4.5 Conclusion

In this chapter we showed three ways of combining decision procedures based on the *Nelson and Oppen* combination framework, and an original alternative that can be seen like an extension, using *model-equalities*.

The propagation by exchanging equalities works very well when theories are convex. When it gets to non-convex theories, disjunction of equalities are not a practical alternative, while exploring all arrangements can make good use of the SAT-solver to obtain good results. Our original work with model-equalities can make the exploration of the arrangements more efficient by building them step by step with the help of the decision procedures and models.

There is still some guessing involved when using model-equalities, but stating theory inconsistent facts is always avoided. Model-equalities are also a good alternative for decision procedures that are not able to deduce all equalities. It differs from [25] in the fact that model-based guessing is now integrated in a classical Nelson-Oppen equality exchange, seeing it just as a new way to exchange equalities.

So far, we have not seen the details of the decision procedures. We did some abstractions and use the decision procedures mainly as black boxes. We will present the details in the next chapter, where we will analyze the requirements to build decision procedures that can be used in SMT-solvers. Later, in Chapters 6 and 7, we will present the specific details to build a decision procedure for difference logic and linear arithmetic.

Chapter 5

Extending a basic decision procedure

A decision procedure¹ is a method to solve a decision problem, where the expected answer is *yes* or *no*. In our case, a decision procedure will decide if an arithmetic formula is satisfiable or not, i.e, if there exists an interpretation of variables that makes the formula true.

However, a decision procedure needs to provide more than just a *yes* or *no* answer if we want to use it in an SMT-solver. This is due to the complex cooperation between decision procedures and also between the decision procedures and the SAT-solver. We have seen many insights in Chapters 3 and 4.

In previous chapters, we looked at decision procedures as black boxes that could give us the *yes* or *no* answer, and also any other extra functionality that we thought necessary. In this chapter we will explain in details these extra requirements.

5.1 Introduction

Decision procedures for SMT-solvers requires extra functionalities. Some of the requirements are important for allowing the SMT-solver to be complete² under some (combination of) theories. Others are important for efficiency, that is strongly related to the capability of handling bigger and more complex formulas. Here we summarize them before explaining the details in the following sections.

1. Conflict set: explanation of a *no* answer.

¹In this thesis decision procedures (DP), theory solvers (\mathcal{T} -solvers) and theory reasoners have similar meanings.

²For every problem the algorithm should eventually terminate with a sound result.

2. Equality generation: deduction of equality between variables and the explanation of why such variables are equal.
3. Model-equality generation: extraction of a model and generation of equalities from variables that have the same value in the model.
4. Lemmas: any valid formula that can be appended to the original formula.
5. Theory propagation: information that can be deduced from the set of known literals.
6. Incrementality: capability of receiving more information at any state and be able to continue without restarting from scratch.
7. Backtrackability: capability of going back to some point in the past, recovering the previous state, and allowing to continue receiving more information.

5.2 Conflict set generation

Given a set of constraints, if the decision procedure detects that the problem is unsatisfiable, we call *conflict set* any subset of these constraints that is also unsatisfiable. The conflict set can be the whole set of constraints, but generally it is better when it is smaller. Moreover, the conflict set is mostly interesting when it is a minimal subset, i.e., when no constraints can be removed without making the subset satisfiable.

As a simple example, given the following set of constraints,

$$\{x - y \leq 0, y - z \leq 1, z - x \leq -1, y - x \leq -1\} \quad (5.1)$$

a decision procedure should be able to detect that the problem is unsatisfiable due to the (minimal) conflict set:

$$\{x - y \leq 0, y - x \leq -1\} \quad (5.2)$$

Minimal conflict sets are important for automatic solvers. They allow the solvers to prune the search space, speeding up the discovery of the final solution. Pruning may work for any general solver and specially well for a DPLL based solver, like we saw in the chapter about SAT-solvers (Chapter 3).

To illustrate this, consider the Formula 5.3. It is a formula in CNF³ as we are used to see in DPLL based solvers. The formula is unsatisfiable and

³Conjunctive Normal Form, i.e., conjunction of disjunctions

the total number of (naive) checks⁴ for concluding this is four. We can see a possible static search tree⁵ representing this formula in Figure 5.1.

$$x > y \wedge y > x \wedge (z = x \vee z = y) \wedge (z = y + x \vee z = 0) \quad (5.3)$$

However, a solver that uses the minimal conflict set for this case, $(x > y) \wedge (y > x)$, will be able to conclude the unsatisfiability with only one check. Simply adding the negation of this minimal conflict set to Formula 5.3 would make the problem propositionally⁶ unsatisfiable⁷.

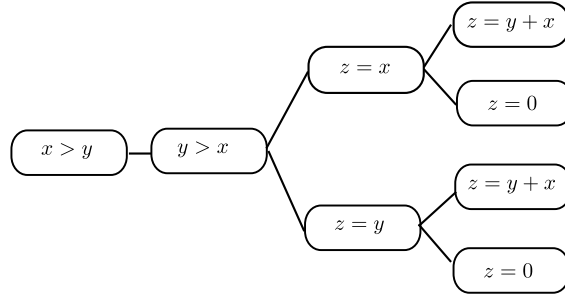


Figure 5.1: Static search tree for the Formula 5.3. A check is represented by the nodes along a path going from the root to one of the leaves. For this tree, 4 checks are possible.

There may be several minimal conflict sets for the same problem. The smallest is not necessary the best. Imagine two minimal conflict sets with no variables in common. They will prune two independent subtrees of the search space and the size of the subtrees is arbitrary.

A conflict set may also contain the explanation of how rules were applied to get to the status of unsatisfiability. We will call this explanation a *certificate*. The rules, explanation and how detailed both of them are will vary from decision procedure to decision procedure. The certificate could be replayed later by an external prover for certifying that nothing is wrong. This gives more credibility to the solver. Users can also stop using the solver as an oracle and verify that the reasoning is correct when they want to.

To illustrate it, consider the Example 5.1. Giving names to the constraints we have:

$$l_1 : x - y \leq 0; l_2 : y - z \leq 1; l_3 : z - x \leq -1; l_4 : y - x \leq -1 \quad (5.4)$$

⁴A check verifies that a propositionally satisfiable assignment is also theory consistent.

⁵Here we say it is static because it is built at the beginning, using the original unchanged formula. One same formula can result in different search trees. It will depend on the order the clauses and literals are analyzed. The paths in different search trees will be the same unless techniques for pruning is applied.

⁶It is not necessary to look for theory inconsistency as there are no more propositional assignments that makes the formula true.

⁷ $(x > y) \wedge (y > x) \wedge \phi \wedge \neg((x > y) \wedge (y > x))$ is unsatisfiable.

The certificate could be simply $l_1 + l_4$ ⁸. Now one could find the contradiction by just replaying the certificate, in this case, adding literal l_1 with l_4 .

(Minimal) conflict sets, with or without certificates, are also useful for the users. They can use them to understand if there is something wrong, so they can reformulate the problem when necessary or conclude that the result is as expected.

5.3 Equality generation

One of the main assumptions of the Nelson and Oppen combination framework is the equality propagation between decision procedures. Every decision procedure in the framework should be able to detect equalities, so these equalities may be propagated to the other decision procedures. That will allow the solver to reason about combined theories.

Without equality propagation, the isolated decision procedures may not have enough information to conclude the unsatisfiability of some combination of theories. For example, given the formula ϕ :

$$\phi : x - y \leq 0 \wedge y - z \leq 1 \wedge z - x \leq -1 \wedge x - w \leq -1 \wedge f(x) \neq f(y) \quad (5.5)$$

The Nelson and Oppen scheme works by splitting the formula and sending constraints to two decision procedures, one to deal with the arithmetic theory and another one to deal with uninterpreted functions. A possible scenario is shown in Table 5.1.

level	DP for arithmetic	DP for uninterpreted functions
0	$x - y \leq 0$ $y - z \leq 1$ $z - x \leq -1$ $x - w \leq -1$ $x = y$ (<i>detected</i>)	$\underline{f(x) \neq f(y)}$
1		$\underline{x = y}$ (<i>new</i>) $\underline{unsatisfiable}$

Table 5.1: Example of equality generation. Underlining is used to emphasize the constraints involved in the conflict.

The Table 5.1 shows that, at the level 0, working alone, the decision procedures cannot detect unsatisfiability. However, an equality is generated and propagated to the other decision procedure. This new information allows the decision procedure for uninterpreted function to conclude that the set of literals is unsatisfiable.

⁸ $l_1 + l_4 \equiv (x - y \leq 0) + (y - x \leq -1) \implies x - y + y - x \leq 0 - 1 \implies 0 \leq -1.$

Once unsatisfiability is detected, we need to build a conflict set to give back to the SAT-solver and allow the process to continue looking for a satisfiable assignment. However, if the conflict contains an equality that was generated by a decision procedure, this equality should not simply be included in the conflict set, since the equality is an internal information that did not exist in the assignment and it may not exist in the formula.

Instead, we add to the conflict set the original constraints that were used to generate the equality. In the example the conflict set would be $x - y \leq 0, y - z \leq 1, z - x \leq -1, f(x) \neq f(y)$. This avoids creating new literals⁹ and allows solving the problem more directly, discarding the previous assignment in the next iteration. When we add the conflict clause to ϕ , we have¹⁰:

$$\begin{aligned} \phi' : x - y \leq 0 \wedge y - z \leq 1 \wedge z - x \leq -1 \wedge x - w \leq -1 \wedge f(x) \neq f(y) \\ \wedge (x - y > 0 \vee y - z > 1 \vee z - x > -1 \vee f(x) = f(y)) \end{aligned} \quad (5.6)$$

ϕ' does not have any more propositional assignments. Therefore we conclude that ϕ is unsatisfiable at this point.

Putting the equality in the conflict set may create a new literal. We would still be able to solve the problem but not in a direct way. In this case we would first generate the conflict clause $f(x) = f(y) \vee x \neq y$. We would get:

$$\begin{aligned} \phi' : x - y \leq 0 \wedge y - z \leq 1 \wedge z - x \leq -1 \wedge x - w \leq -1 \wedge f(x) \neq f(y) \\ \wedge (f(x) = f(y) \vee x \neq y) \end{aligned} \quad (5.7)$$

As a new literal is introduced in this clause, the next assignment would be the same, except for $x \neq y$ that will be included. In the next iteration the conflict set $x - y \leq 0, y - z \leq 1, z - x \leq -1, x \neq y$ would be created. Adding it to ϕ' would finally make the problem propositionally unsatisfiable.

$$\begin{aligned} \phi'' : x - y \leq 0 \wedge y - z \leq 1 \wedge z - x \leq -1 \wedge x - w \leq -1 \wedge f(x) \neq f(y) \\ \wedge (f(x) = f(y) \vee x \neq y) \\ \wedge (x - y > 0 \vee y - z > 1 \vee z - x > -1 \vee x = y) \end{aligned} \quad (5.8)$$

5.4 Model-equality generation

When integer variables are present, the linear arithmetic theory is no longer convex. A theory is non-convex when disjunction of literals can be deduced but none of the literals alone can be, e.g., a decision procedure cannot affirm that a constraint l_1 is true, neither that a constraint l_2 is true, but it can

⁹Creating new literals may increase the search tree, possibly making us to test more assignments.

¹⁰We represent $\neg(f(x) \neq f(y)) \equiv f(x) = f(y)$, $\neg(x - y \leq 0) \equiv x - y > 0$, ...

deduce that $l_1 \vee l_2$ is true. Examples of non-convex theories are integer difference logic and linear integer arithmetic.

To illustrate the non-convexity of integer arithmetic, consider this simple example.

$$x \geq 0 \wedge x \leq 1 \tag{5.9}$$

One cannot deduce that $x = 0$. Neither that $x = 1$. However, we can state that $x = 0 \vee x = 1$.

For such non-convex theories, just propagating the simple detected equalities, as it was done previously, is not enough for completeness. One alternative proposed by Nelson and Oppen is to generate disjunction of equalities, but it does not work in practice because of the difficulties of finding these disjunctions. A better alternative that we will follow is to generate equalities from a model, like we explained in Chapter 4.

To briefly remind how the process works, take a look at the following example. It contains a set of arithmetic constraints.

$$x \geq 0 \wedge x \leq 1 \wedge y = x + 1 \wedge z = 2 \tag{5.10}$$

In a model, each variable has a value associated to it. The two possible models for this case are:

$$x = 0; y = 1; z = 2 \tag{5.11}$$

$$x = 1; y = 2; z = 2 \tag{5.12}$$

Out of the model in 5.12, we could extract the equality $y = z$. The equality is not always true, but using it as a *model-equality* allows us to achieve completeness. We just need to propagate the model-equalities like normal equalities. By interacting with the SAT-solver, if there is a conflict with a model, we can backtrack and try different ones until one of them is accepted as valid or all of them are tried and stated as conflicting.

One important difference to normal equalities is that we do not have an explanation for how the model-equalities were created. That is because a model-equality is some kind of guess and not a deduced fact. Model equalities will appear in conflict sets. This will possibly create and introduce new literals in the formula of the SAT-solver, but the solver still can solve the problem with extra interactions by performing like in the second part of the Example 5.6 from the previous section. For a detailed explanation of model-equalities, see Section 4.2.

5.5 Lemmas

In this framework, lemmas are a generic way of giving some fresh information to the SAT-solver. So it can be considered as a way of interaction between

the decision procedures (or any other component of the SMT-solver) and the SAT-solver.

Any tautology can be given to the SAT-solver as a lemma. But lemmas will be interesting when they can speed up the process or help the decision procedures to achieve completeness.

The speed up can be obtained by creating lemmas that will prune the search space in the SAT-solver. An example is theory propagation lemmas, that will be presented in Section 5.6. Conflict sets can also be considered lemmas. The closer to a minimal subset the conflict sets are, the better the gain with the performance we can obtain, as the information is more precise.

As an example of use of lemmas to achieve completeness, imagine one arithmetic decision procedure that cannot handle disequalities (\neq). It can use lemmas to obtain some information that it can understand in the next interaction with the SAT-solver. So if it receives $x \neq y$ it can generate a lemma like $x \neq y \implies x > y \vee x < y$. A possible advantage of using lemmas, is that they could be generated in a lazy way, just when/if necessary.

It is up to the components of the SMT-solver to decide when they want to generate lemmas. However, they should be careful to not pollute or overpopulate the number of clauses in the SAT-solver, as this can have a negative influence over the performance.

5.6 Theory propagation

One technique that SMT-solvers can use to improve performance is theory propagation (first presented by Nieuwenhuis and Oliveras [55]). The classical DPLL algorithm only deduces facts using propositional logic. Theory propagation also reasons about theories to deduce facts.

The idea behind theory propagation is to use a *theory reasoner* (i.e. a decision procedure) for guidance, instead of blindly deciding literals in the SAT-solver. A theory reasoner could tell the SAT-solver which literals are true among the current set of literals under the current assumptions. Therefore, the SAT-solver can directly set these literals to true, not having to decide (or guess) if they are true or false. In this way, the search space of the SAT-solver can be reduced significantly.

One thing we should take in consideration is that from a set of constraints, possibly an infinite number of others can be deduced. Take for instance $x > 0$. If we know that x is greater than 0, then we can easily affirm that x is greater than -1 , -2 , -3 , ... It will be only interesting to the solver to deduce the ones that are relevant to the problem, i.e., to deduce literals that are also in the formula. Therefore, somehow, the theory reasoner should be aware of all the literals from the problem in advance.

To illustrate theory propagation, consider the following example. We

want to find a satisfiable assignment for ϕ .

$$\phi : z = 1 \wedge y \leq 0 \wedge x \geq 2 \wedge (x \geq 1 \vee y \geq 0) \wedge (x - y \leq 0 \vee x = y \vee z + y \leq 1) \quad (5.13)$$

Looking at a static search tree for ϕ , we see 6 possible assignments in which 4 are unsatisfiable. They are shown in Figure 5.2.

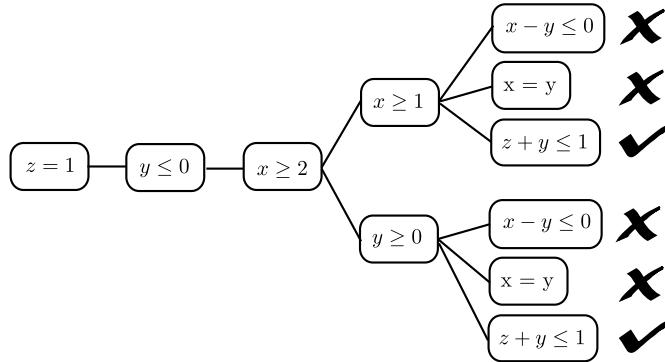


Figure 5.2: Static search tree for the Formula 5.13.

Using theory propagation, literals can be deduced, like $(z = 1 \wedge y \leq 0 \implies z + y \leq 1)$ or $(x \geq 2 \implies x \geq 1)$. Deducing literals can reduce the number of guesses the SAT-solver has to do. In this example, if we use theory propagation, we can find a satisfiable assignment directly¹¹.

$$\phi : \underbrace{z = 1 \wedge y \leq 0 \wedge x \geq 2}_{\text{deduced}} \wedge \underbrace{(x \geq 1 \vee y \geq 0)}_{\text{deduced}} \wedge (x - y \leq 0 \vee x = y \vee \underbrace{z + y \leq 1}_{\text{deduced}})$$

Figure 5.3: Theory propagation: literals can be deduced from others.

Conflict sets can also be seen as some kind of theory propagation, a lazy one. When a theory solver finds a problem with the current assignment, it returns back a conflict set. This conflict set is a new clause that is incorporated into the formula and will force the SAT-solver not to take the same wrong decision again.

In Figure 5.4, we can see that $y \leq 0 \wedge x \geq 2$ implies that x and y are different. Incorporating this information in the formula would reduce the search space by eliminating all the assignments that contain $y \leq 0 \wedge x \geq 2 \wedge x = y$.

As a final remark, notice that theory propagation is very interesting when it is inexpensive¹² to ask for deduced information. When cost increases, it

¹¹The assignment that is built guided by theory propagation has no inconsistency.

¹²Preferably constant time or very close to.

$$\phi : z = 1 \wedge \underline{y \leq 0 \wedge x \geq 2} \wedge (x \geq 1 \vee y \geq 0) \wedge (x - y \leq 0 \vee x = y \vee z + y \leq 1)$$

Figure 5.4: Conflict sets can be seen as theory propagation. Here $y \leq 0 \wedge x \geq 1 \implies x \neq y$.

may not be a good idea to ask for such information at every iteration. Having a partial¹³, but faster theory propagation may be also interesting. Doing full or partial propagation, and all the time or once in a while, may influence the performance of the solver [30]. Experimenting different strategies for different benchmarks may yield quite different results.

5.7 Incrementality and backtrackability

Incrementality is the capability of receiving more information at any state and be able to continue without restarting from scratch. For the decision procedures, incrementality is important because they very often receive new constraints that should be incorporated and processed with the old ones.

Incrementality comes very often with backtrackability. Backtrackability is the capability of going back to some point in the past, recovering the previous state, and allowing to continue receiving more information. Both incrementality and backtrackability should be efficient in terms of space and time complexity.

There are several reasons for including an incremental and backtrackable decision procedure:

1. Avoid unnecessary work. Decision procedures work with assignments given by the SAT-solver that may contain lots of constraints. If a small subset of these constraints makes the problem unsatisfiable, there is no reason to continue working as the current assignment is also unsatisfiable. So the sooner we detect the unsatisfiability of the assignment, the better it is for avoiding unnecessary processing of the remaining constraints.
2. There is usually a very strong interaction with the SAT-solver. Since for most problems, the unsatisfiability can be detected early, it is a smart decision to test partial assignments to see if they are already unsatisfiable. A lot of time can be saved avoiding to build complete assignments in the cases that a partial assignment is already conflicting. It is also interesting for pruning the search tree more efficiently.

¹³We do not need to find all logical consequences from the current set of constraints that we have. We only look for logical consequences that are cheap to find.

Checking partial assignments means receiving the constraints by small blocks, and it is there where incrementality plays an important role.

3. Consecutive set of constraints given to the decision procedures are usually very similar to each other. The SMT-solver may check for theory consistency of several assignments until it finds a satisfiable assignment or try them all and find out that the formula is unsatisfiable. Due to the nature of the DPLL algorithm in the SAT-solver, consecutive assignments are usually very similar to each other. If there are literals in consecutive assignments that are the same, there is no reason for removing them all from a decision procedure to later give them all back to it. So smart SMT-solvers will backtrack to a common point of consecutive assignments and then just start giving the new literals to the decision procedure. So in this case incrementality is important as well as being backtrackable.
4. Once all the literals are given from the SAT-solver to a decision procedure, extra constraints, notably equalities and model equalities, can still be propagated between the decision procedures. The reason is that SMT-solvers are based on the Nelson and Oppen framework. This propagation process can last for some iterations. Each time new constraints are given, decision procedures should process them incrementally for efficiency reasons. There should be no distinction in the way decision procedures handle incrementality, i.e., the decision procedure does not need to know if the new constraint comes from another decision procedure or from the (new assignment of the) SAT-solver.

For illustrating the first three reasons, consider the following example. Consider the propositional formula:

$$A \wedge B \wedge (C \vee \neg D) \wedge (D \vee E \vee F) \wedge (\neg E \vee \neg F) \quad (5.14)$$

We will look only at propositional level for simplicity. From Equation 5.14 we extract the static search tree shown in Figure 5.5.

There are 6 propositional satisfiable assignments from the formula that are shown in Figure 5.5. They are all partial, meaning that there is at least one literal that was not set to either true or false. If we consider the total assignments, there are 12 propositionally satisfiable that can be seen in Figure 5.6.

When working with partial assignments, if a conflict is found by a decision procedure during a consistency check, additionally to stop the work earlier, we avoid exploring a subtree possibly containing several other assignments. Imagine that at some point the SMT-solver checks for the theory consistency of the partial assignment $\Gamma = \{A, B, C\}$ and then it finds out that the assignment is theory inconsistent. If the SMT-solver learns that

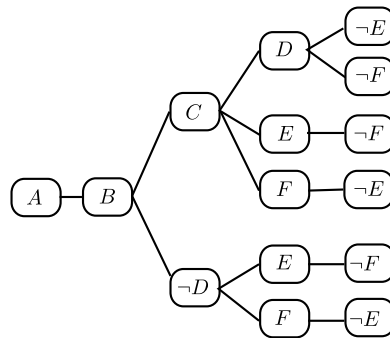


Figure 5.5: Static search tree reflecting propositional satisfiable assignments from Equation 5.14.

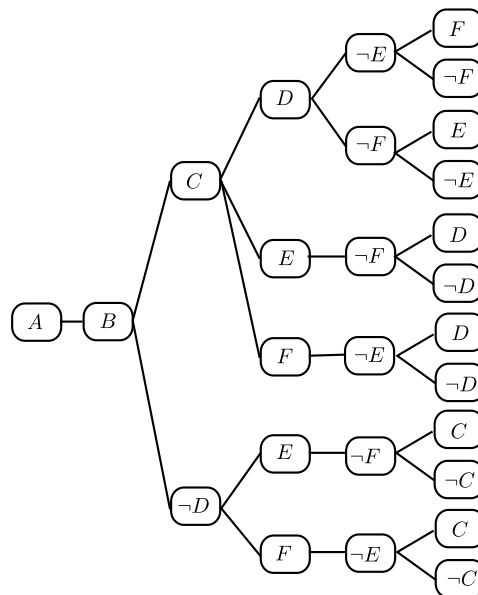


Figure 5.6: Static search tree reflecting the propositional satisfiable total assignments of Equation 5.14.

$A \wedge B \wedge C$ is false¹⁴ and uses this information, it can prune more than half of the tree, avoiding plenty of unnecessary checks.

Now imagine another scenario with the same formula where all the assignments are unsatisfiable and the reason is always the entire set of the literals from each assignment. You can notice that when checking $\Gamma_1 = \{A, B, C, D, E\}$, $\Gamma_2 = \{A, B, C, D, \neg F\}$, $\Gamma_3 = \{A, B, C, E, \neg F\}$, ...

¹⁴Meaning that $\neg A \vee \neg B \vee \neg C$ is true.

an important subset of the assignment is the same. Going from one to the other, backtracking to a common point and then restarting incrementally is crucial to avoid reprocessing all the common literals of the assignments.

To illustrate the fourth reason, we take as example the Formula 5.15, that has been already purified. We look for unsatisfiability using classical Nelson and Oppen, like explained in Section 4.1, using three decision procedures: one for arithmetic (DP A); one for uninterpreted function (DP UF); and one for lists (DP L).

$$\begin{aligned}
 & x \leq y \wedge y \leq v_1 \wedge v_2 = v_3 - v_4 \wedge v_5 = 0 \\
 \wedge P(v_2) = true \wedge P(v_5) = false \wedge v_3 = f(x) \wedge v_4 = f(y) \\
 & \wedge v_1 = car(cons(x, \ell)) \quad (5.15)
 \end{aligned}$$

We are checking for satisfiability of the full assignment. None of the decision procedures detects the unsatisfiability by itself, but when they propagate new information they have deduced, the unsatisfiability is found. Observe that equalities are detected and propagated a few times. Each time, the other decision procedures receive the new constraint and perform a new consistency check. Here we once again see the importance of incrementality, the constraint set the decision procedures are handling are the same except for the new deduced equality.

level	DP A	DP UF	DP L
0	$x \leq y$ $y \leq v_1$ $v_2 = v_3 - v_4$ $v_5 = 0$	$P(v_2) = true$ $\underline{P(v_5) = false}$ $v_3 = f(x)$ $v_4 = f(y)$	$v_1 = car(cons(x, \ell))$ $v_1 = x$ (detected)
1	$v_1 = x$ (new) $x = y$ (detected)	$v_1 = x$ (new)	
2		$x = y$ (new) $v_3 = v_4$ (detected)	$x = y$ (new)
3	$v_3 = v_4$ (new) $v_2 = v_5$ (detected)		$v_3 = v_4$ (new)
4		$v_2 = v_5$ (new) <u>unsatisfiable</u>	$v_2 = v_5$ (new)

Table 5.2: Importance of incrementality in a Nelson and Oppen framework. Equalities are deduced and propagated to the decision procedures until, in this example, unsatisfiability is found. Underlining is used to emphasize the constraints involved in the conflict.

5.8 Conclusion

As we have seen, building efficient decision procedures for SMT-solvers requires much more than simply *yes* or *no* answers. In this chapter, it was presented the details of these extra requirements describing exactly what we expect from the decision procedures. Most of the motivations of these extra requirements come from the SAT-solver and the combination framework based on Nelson and Oppen, that we have seen in previous chapters.

In the next chapters, we present how to build decision procedures for fragments of arithmetic theory, fulfilling all these requirements. A first decision procedure will be given for the low complexity fragment of difference logic, followed by linear arithmetic.

Chapter 6

Deciding difference logic

Arithmetic is a very large theory. Problems involving arithmetic usually contain only a subset of functions and symbols. Therefore, most of the time we can limit ourselves to interpreting the problem in a well known fragment of arithmetic. Several fragments have been extensively studied. There are many decision procedures for them that have been constantly used and extended for different areas.

In this chapter we present the arithmetic fragment of difference logic. It is a decidable fragment with very low complexity. It can be used to solve problems in many different areas.

We start by presenting the difference logic theory using graphs to model the problems. Then we show the details to build a decision procedure for difference logic.

6.1 Difference logic graph theory

In this section, we describe difference logic. We give the details of how it can be understood using graph theory and show some properties necessary for understanding the algorithms for the decision procedures that will be presented in Section 6.2.

Difference Logic (DL) is the fragment of arithmetic that handles constraints of the type $x - y \leq c$, where x and y are variables and c is a numerical constant, and they can be integer or rational. It appears in many different and important practical problems such as timed systems, scheduling problems, paths in digital circuits, see, e.g., [54]. They also are the predominant kind of constraint in many problems involving arithmetic. One can see many examples of industrial problems in the benchmarks of the SMT-LIB [59].

Difference logic is a well studied subject and it can be fully modeled using graph theory. Therefore, solvers that are aware of these facts, can make use of fast algorithms that are designed for graph theory and may accomplish good performances. In this section, we focus on the description

and illustration of interesting properties necessary to our decision procedure.

6.1.1 Properties and Graph Representation

The classic difference logic problem deals only with constraints of the kind $x - y \leq c$. It can be interpreted in graph theory as an edge from y to x with cost (or weight) c , see Figure 6.1, for both real and integer theory. That can be read as node (or vertex) x should be at most node $y + c$.

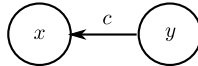


Figure 6.1: Representation of the constraint $x - y \leq c$ using graphs.

There are other constraints that can be easily translated and integrated to this graph model. Table 6.1 gives a few of the common constraints that can be translated to DL.

constraint	translated to
$x - y \geq c$	$y - x \leq -c$
$x - y = c$	$x - y \leq c$ and $y - x \leq -c$
$x \leq y$	$x - y \leq 0$
$x \geq y$	$y - x \leq 0$
$x = y$	$x - y \leq 0$ and $y - x \leq 0$
$x \leq c$	$x - v_0 \leq c$, where v_0 is a unique extra variable with value 0
$x \geq c$	$x - v_0 \geq c$, where v_0 is a unique extra variable with value 0

Table 6.1: Table of constraints

Strict inequalities can also be handled with minor changes. The method shown in this section is presented here [31] and is used by many arithmetic decision procedures.

We can always change a constraint like $x - y < c$ to $x - y \leq c - \delta$. The value of δ depends on the type of numerical variables presented in the constraint. If they are all integers, δ is precisely 1, e.g., we can change $x - y < 1$ to $x - y \leq 0$ without any loss of precision. But if there are rationals or reals in the constraint, δ has to be a very small value, small enough to not change the result of evaluating the constraints.

It is a hard task to determine the value of δ in the general case. However, we can always say that the value of δ is infinitely small non zero value. Representing this value in the computer may not be possible, but we do not actually need it. We can say that δ is $\frac{1}{\infty}$, do the calculations using the

symbol δ as a variable. The decision procedures will never need to evaluate the real value of δ .

Instead of evaluating a number c as itself, we think of it as a pair (c, k) equivalent to $c + k\delta$. For example, translating $x - y < c$ results in $x - y \leq (c, 1)$, while translating $x - y \leq c$ result in $x - y \leq (c, 0)$. The operations on the pair are like the ones we use for equations $(c + k\delta)$, where c and k are known and δ is a variable. Only a few operations will be necessary:

- $(c_1, k_1) + (c_2, k_2) \equiv (c_1 + c_2, k_1 + k_2)$
- $(c_1, k_1) - (c_2, k_2) \equiv (c_1 - c_2, k_1 - k_2)$
- $c' \times (c, k) \equiv (c' \times c, c' \times k)$

Additionally, we can also compare two pairs because we know the value of δ . We know that:

- $(c_1, k_1) \leq (c_2, k_2) \equiv (c_1 < c_2) \vee (c_1 = c_2 \wedge k_1 \leq k_2)$

After building a graph with the entire set of difference logic constraints we can observe many interesting facts. The following subsections will describe the interesting ones for the case of satisfiability.

Dependency

If two variables x and y do not depend on each other, there will not be a path from x to y , nor a path from y to x . In graph theory, a path from x to y is a sequence of contiguous edges that connect x to y .

A variable x may depend on another variable y either directly or indirectly. In the first case there will be an edge from x to y and thus a path from x to y . In the second case, there will be a path from x to y that represents a combination of the constraints in the path and the length (in our case, the sum of the edges costs) indicates the relationship between the variables.

For instance, take two constraints $x - y \leq -1$ and $y - z \leq -2$. The combination of them ($x - z \leq -3$) shows the indirect dependency between x and z . The graph representation for this example can be seen in Figure 6.2.

Strongest constraint

As the name suggests, it is the strongest constraint that can be created (or extracted) from a set of constraints, related to a pair of variables. If we find that $y - x \leq c_1$ is the strongest constraint related to y and x , that means that we cannot extract any other constraint $y - x \leq c_2$ where $c_2 < c_1$.

If there is a path from x to y , the shortest path len from x to y gives the strongest constraint ($y - x \leq len$) that it is possible to create from the given

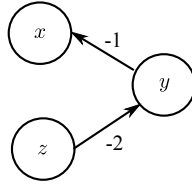


Figure 6.2: Direct ($x - y \leq -1$ and $y - z \leq -2$) and indirect ($x - z \leq -3$) dependency of variables.

facts. Notice that the shortest path in our case is related to the sum of the edges costs and not to the number of the edges. One can also interpret it as y must be at most $x + len$.

For instance, the following set of constraints is represented by the graph in Figure 6.3. They are

$$y - x \leq -1, z - x \leq -2, w - y \leq -3, w - z \leq 0, y - z \leq 0.$$

The strongest constraint that can be built between x and w is $w - x \leq -5$, built from the shortest path from x to w (that goes through z and y).

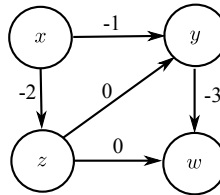


Figure 6.3: The strongest constraint related to two variables (vertices) can be built from the shortest path between them. In the figure, we have three strongest constraints ($w - x \leq -5$, $w - z \leq -3$ and $y - x \leq -2$) that are derived from the shortest path and are not given in the set of original constraints.

Unsatisfiability

A set of constraints is unsatisfiable when it is possible to find a contradictory subset of constraints like $x - y \leq -1 \wedge x - y \geq 2$. If there is no combination of constraints that can make the problem unsatisfiable, the problem is satisfiable.

Theorem 4. *There is a negative cycle in the graph if and only if the problem is unsatisfiable.*

Proof. If there is a negative cycle in the graph then there is a path from a vertex x to itself with negative weight. Therefore, we can get the implied constraint $x - x \leq c$, for some $c < 0$, which is a contradiction.

For the other half, in a system of inequalities of the kind $x - y \leq c$, there is a contradiction only if there is a subset of the inequalities that combined lead to the inequality $0 \leq c$, for some $c < 0$. To reach this contradiction, the variables should appear at least twice in the inequalities and the sum of the coefficients of all variables should be 0. Based on these facts, for any variable x present in this subset, the implied inequality $x - x \leq c$, for some $c < 0$, is valid. In the presented graph model, this means there is a path from x to x with negative weight $c < 0$, i.e., a negative cycle. \square

The check for satisfiability is done by looking for negative cycles. In case no negative cycle is detected the problem is satisfiable.

Figure 6.4 shows a scenario where there is a negative cycle. The proof for the unsatisfiability (or conflict set) can be constructed from the edges in the negative cycle. The combination of these constraints will lead to the contradiction $0 \leq -1$.

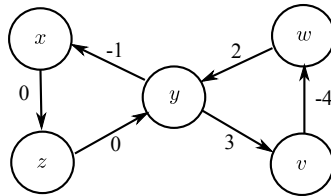


Figure 6.4: Graph with a negative cycle. The set of constraints representing it is unsatisfiable. The proof is the edges from the negative cycle: $z - x \leq 0$, $y - z \leq 0$ and $x - y \leq -1$. If we combine these constraints, by simply summing them up, we obtain $0 \leq -1$.

There can be many negative cycles in a graph. Efficient algorithms for finding negative cycles are not interested in finding the smallest one. But any of the negative cycles used in a proof of unsatisfiability are minimal. That means, if any of the constraints in the cycle is removed, the remaining subset of constraints is not unsatisfiable anymore.

Equality between variables

Theorem 5. *In a satisfiable problem, two variables x and y are equal if and only if the shortest path from x to y costs 0 and the shortest path from y to x also costs 0.*

Proof. The (strongest) constraints representing these two shortest paths are $x - y \leq 0$ and $y - x \leq 0$, and thus $x - y = 0$ (or $x = y$).

For the other half of the theorem, if we have two variables that are equal, x and y , we have: $(x = y) \implies (x - y = 0) \implies (x - y \leq 0 \wedge x - y \geq 0) \implies (x - y \leq 0 \wedge y - x \leq 0)$. The later means there is a path from x to y and from y to x with cost zero that can be either an original constraint of the problem, or an implied constraint. In either way, they cannot be *stronger* because otherwise the problem would be unsatisfiable. \square

Figure 6.5 shows an example where an equality is found using shortest path information.

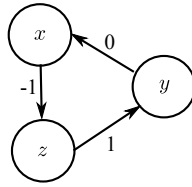


Figure 6.5: Graph made from 3 constraints: $z - x \leq -1$, $y - z \leq 1$ and $x - y \leq 0$. An equality can be found in this graph: $x = y$.

$x - y \leq 0 \wedge y - x \leq 0 \implies x = y$ is exactly the way back from understanding a constraint like $x = y$ shown in Table 6.1. The difference is that this information might be obtained indirectly and not as given constraints.

6.1.2 Conclusion

We presented difference logic and a way of representing this theory using graphs. We showed some interesting properties that make easier to understand a few algorithms for the decision procedures. We know, among other things, that we can check for satisfiability using algorithms for negative cycle detection. Additionally, the theorem of equality between variables will give us enough knowledge to present an algorithm for finding equalities. In the next section, we will use what we learn here to build a decision procedure for difference logic.

6.2 Difference logic decision procedure

The previous section introduced the difference logic arithmetic fragment. We saw some interesting properties and how to interpret this theory using graphs.

In this section, we present a series of algorithms that when put all together can be used to build a decision procedure for difference logic. Many of the following algorithms reflect our implementation in the `veriT` solver. We show the strong and weak points of our decision procedure, and present some alternatives for fulfilling the requirements.

6.2.1 Satisfiability Checking

The first basic requirement for a decision procedure is be able to determine if a set of constraints is satisfiable or not. We saw in Section 6.1.1 that a set of constraints is unsatisfiable if and only if the graph built from the constraints has a negative cycle.

The classical algorithm to check if a graph has a negative cycle is the Bellman-Ford algorithm [8, 34]. The original algorithm is not incremental. Our later proposed algorithm does not follow the same idea and is not based on the Bellman-Ford algorithm. But the Bellman-Ford algorithm is the most widely used algorithm for detecting negative cycles, so we show it in this section for reference. A pseudo-code can be seen in Algorithm 4.

```

input :  $G=(V,E)$ : Graph
input : source: Vertex  $\in V$ 
output: hasNegativeCycle: Boolean
data  :  $u, v$ : Vertex

// Initialize graph
1 foreach Vertex  $v$  in  $V$  do
2   |  $v.distance := \infty$ ;
3   |  $v.predecessor := NULL$  ;
4 end
5 source.distance := 0;

// Relax edges repeatedly
6 for  $i = 1$  to  $V.Size()$  do
7   | foreach Edge  $e$  in  $E$  do
8     |  $u := e.source$  ;
9     |  $v := e.destination$  ;
10    | if  $v.distance > u.distance + e.weight$  then
11      |  $v.distance := u.distance + e.weight$  ;
12      |  $v.predecessor := u$  ;
13      | // Check for negative-weight cycles
14      | if  $i = V.Size()$  then
15        | | return hasNegativeCycle := true ;
16      | end
17    | end
18 end
19 return hasNegativeCycle := false ;

```

Algorithm 4: BellmanFord

The Bellman-Ford algorithm computes single-source shortest paths in a graph. But it can additionally detect negative cycles by including a simple

check after the last iteration. It is a non incremental algorithm and runs in $O(|V||E|)$, where $|V|$ is the number of vertices and $|E|$ is the number of edges.

The idea of the Bellman-Ford algorithm is based on relaxing the estimated distance between the *source* and all the other vertices. After the cycle i from the *for* loop (line 4), $v.distance$ will contain the shortest path from *source* to v using at most i edges. After $|V| - 1$ cycles, the distances from *source* to the vertices should be the shortest path. The exception is when there is a negative cycle reachable from *source*. If we can still reduce the distance by doing an extra iteration, it means there is a negative cycle. In such cases, the shortest path can always be reduced by going through the cycle an infinite number of times.

6.2.2 Incremental Satisfiability Checking

In this subsection we present an original incremental satisfiability check algorithm, Algorithm 5. It can be implemented using a heap with complexity $O(|E'| \log |V'|)$, where $|V'|$ is the number of vertices that have their distance changed in the algorithm and $|E'|$ is the number of outgoing edges of the V' vertices.

Min-heaps are tree based data structures where the root represents always the smallest¹ element of the tree and it has the property that if B is a child of A then $B \geq A$. The common operations of a heap are *find minimal*, *insert*, *delete*, *decrease value* and *merge*. There are several different implementation of heaps with varying complexities. Fibonacci heaps, for instance, have amortized complexity $\Theta(1)$ for all operations, except for *delete* which is $O(\log n)$.

Since the algorithm is incremental, it is executed for each new added constraint. We can do this up to $|E|$ times, stopping as soon as a conflict is found. Another parameter of the complexity depends on the number of vertices that have their distance changed during the algorithm. This is hard to predict. But we know that in general, the stronger the new added constraint is, the greater will be the chance of making relationships between vertices stronger, and so, the higher will be the number of vertices with their distance changed. However, in many cases, where the constraints are weak, or the involved vertices are isolated from the rest of the graph, adding a constraint will take constant time.

The idea of the algorithm is to search in the graph for the vertices that may have their distances changed due to the last added edge. The distance is the length of the shortest path from an arbitrary source vertex. For this purpose and for simplicity, an *artificial vertex* can be created to be the source vertex. It will connect to all the other vertices with a unidirectional edge of

¹Max-heaps can be defined dually.

```

input : G =(V,E): Graph
input : e: Edge
output: hasNegativeCycle: Boolean
data : impr: Number
data : dest, pred, newDest: Vertex
data : q: Heap < Vertex, Vertex, Number >

1 pred := e.source, dest := e.destination ;
2 impr := (pred.distance + e.weight) - dest.distance ;
  // Initial check, improve means decrease the distance
3 if impr < 0 then
4   q := NewHeap();
5   q.InsertOrImprove(dest, pred, impr);
  // Improving Search
6   while not q.Empty() do
7     dest, pred, impr := q.RemoveMin();
8     dest.distance := dest.distance + impr ;
9     dest.predecessor := pred ;
10    foreach OutgoingEdge i in dest do
11      newDest := i.destination ;
12      impr := (dest.distance + i.weight) - newDest.distance ;
13      if q.InsertOrImprove(newDest, dest, impr) = true
14        then
15          if newDest = e.source then
16            return hasNegativeCycle := true ;
17          end
18        end
19    end
20 end
21 return hasNegativeCycle := false ;

```

Algorithm 5: IncrementalNegativeCycleDetection

weight 0. No vertex has edges going to this artificial vertex, so a negative cycle will never be introduced in the graph. Therefore, we can state that the original system will be unsatisfiable if, and only if, the slightly modified one (with the artificial vertex) is.

The algorithm starts by checking if the new edge will improve the distance to the destination vertex. If it does, the search starts. The search is done by greedily picking the vertex in the heap that will have its distance improved the most. The improvement is the difference between the new and old value of distance. It is assumed that, when a vertex v is improved, its

neighbors will have their improvement by at most the same as v . So, when a greedily picked vertex has its distance improved, it is done by the path that will improve the distance by the largest amount. Therefore, each vertex should have their distance improved at most once. The exception is when there is a negative cycle.

A negative cycle happens if and only if the $e.source$ distance improves. Because if $e.source$ improves that means $e.destination$ will improve again, so a cycle is present.

Figures 6.6 to 6.10 illustrate the algorithm. The constraints $\{d - a \leq 1, c - a \leq -1, b - a \leq -1, e - b \leq -1, e - c \leq -2, f - e \leq 4\}$ have been already processed and the figures show behavior of the Algorithm 5 for the addition of the new constraint $a - f \leq -2$. The graph is build as explained in Section 6.1. Additionally, we show the distance information of each variable in the squares and the state of the heap used in the algorithm. The numbers next to the vertices in the heap indicate how much the vertex can be improved taking in consideration all the paths going through the previous selected vertices.

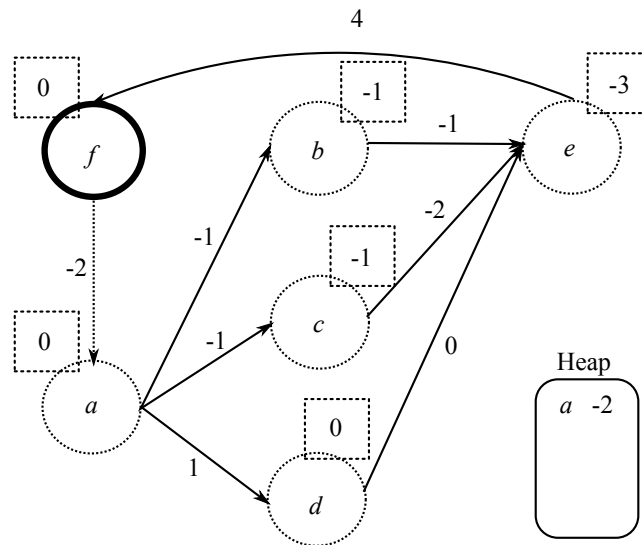


Figure 6.6: Graph representation of an arbitrary example, initial state during the insertion of the edge $a - f \leq -2$. The dashed circles are the vertices that have never been processed, i.e., they have never been chosen from the heap. Bold circle is the vertex that is currently selected. The squares shows the distance from the artificial vertex (not shown here).

At each step, the chosen vertex will have its distance improved. Then it checks the vertices it can reach to verify if their distance can be improved.

When the source vertex f is reached for improvement, a negative cycle

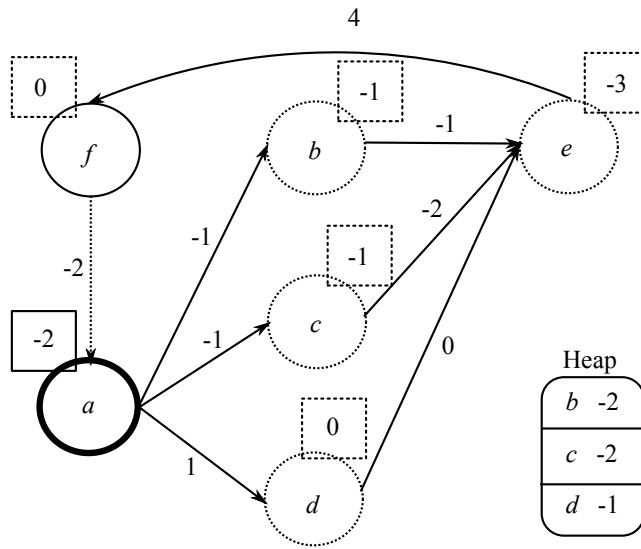


Figure 6.7: Graph representation of the state after the first iteration. a is chosen and adds to the heap vertices b, c and d .

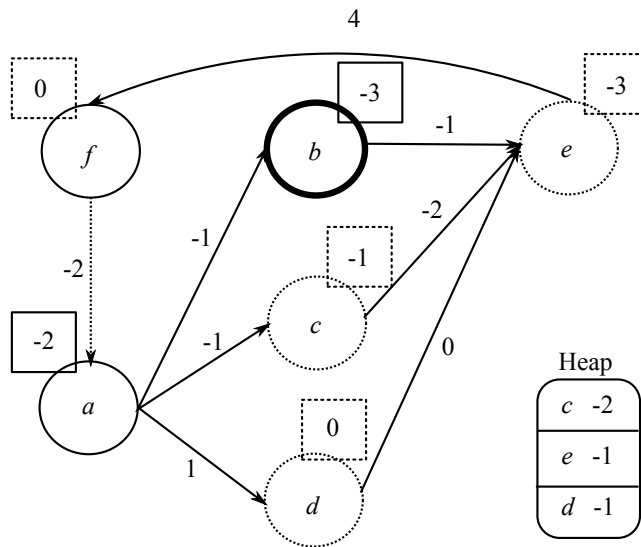


Figure 6.8: Graph representation of the state after the second iteration. b or c could have been chosen, as they are the vertices that can have their distance improved by the largest amount. b is chosen and adds to the heap the vertex e .

is detected. Figure 6.10 shows the final state of this example.

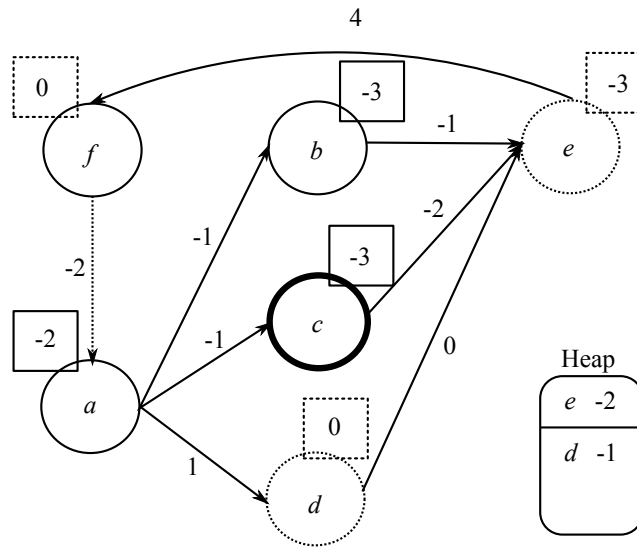


Figure 6.9: Graph representation of the state after the third iteration. c is chosen. e can be reached using a shorter path going through c , so the value of e in the heap is updated.

The execution of the algorithm depends on the values in the edges. A simple modification like changing the order in which edges are added may influence the running time. It is not hard to build an example where the full graph is explored for every new edge added, or the other way around, when all new edges can be added in constant time. The optimal strategy would be adding the edges in an order such that the destination vertex would have no outgoing edges. In practice, the algorithm works very well and it is suitable for sparse and dense problems. One can see some running experiments in [27].

6.2.3 Conflict Set Construction

When unsatisfiability is detected, an explanation is necessary. The explanation is the set of constraints that generated the conflict, as seen in Section 5.2. A minimal conflict set can be obtained by the constraints that form the negative cycle.

The algorithm for checking unsatisfiability keeps the predecessor of the vertices while updating the shortest path distances and doing the search for a negative cycle. So, if a negative cycle is detected, it is only necessary to go through the cycle, by using the predecessor, and collect the constraints associated to the edges. A pseudo-code for this is shown in Algorithm 6.

When a new edge is added, more than one negative cycle may arise. See for instance, Figure 6.11. In this case, it is possible to return more than one

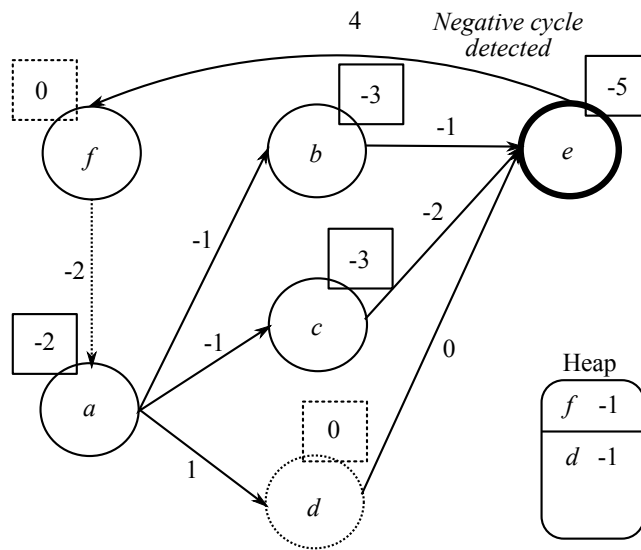


Figure 6.10: Graph representation of the state after the fourth iteration. e is chosen and adds f to the heap. f is the source of the edge that is being added to the graph. As its distance can be improved we know there is a negative cycle in the graph. Therefore, the current set of constraints is unsatisfiable.

```

input :  $G = (V, E)$ : Graph
input :  $e$ : Edge
output:  $s$ : Set of Constraints
data  :  $v$ : Vertex

1  $v := e.source$  ;
2  $s.Insert(e.constraints)$ ;
3 while  $v \neq e.destination$  do
4   |  $s.Insert(Edge(v.predecessor, v).constraints)$ ;
5   |  $v := v.predecessor$  ;
6 end
7 return  $s$  ;

```

Algorithm 6: CollectConflictSet

minimal conflict set. But the algorithm will return the first negative cycle found.

6.2.4 Equality Generation

It was shown in Section 6.1.1 that, in our graph model, two variable x and y are equal if and only if the length of the shortest path between x and y

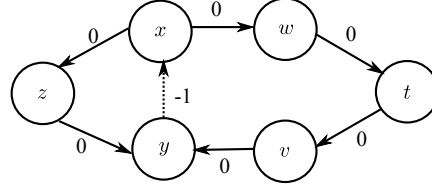


Figure 6.11: In this graph, representing the set of constraints $\{y - z \leq 0, z - x \leq 0, w - x \leq 0, t - w \leq 0, v - t \leq 0, y - v \leq 0\}$, two negative cycles are found after the edge $x - y \leq -1$ is added. The cycles are $\{y, x, z, y\}$ and $\{y, x, w, t, v, y\}$. Any of them can be used to construct the conflict set.

and between y and x are 0.

Algorithm 7 shows how to generate equalities without explicitly calculating the shortest path between each pair of vertices in the graph. It assumes that no negative cycle is present. The algorithm is based on the algorithm by Lahiri and Musuvathi [42].

It is not an incremental algorithm, so it is better to use it at the end after all the constraints have been added to the decision procedure. It can be implemented with complexity $O(|V| \log |V| + |E|)$, where $|E|$ is the number of edges in G and $|V|$ is the number of vertices in G .

The algorithm starts by building the graph G' . That is done by collecting the edges from G where the *slack* is equal to zero. An edge e has *slack*(e) zero when e is part of a shortest path between the *artificial* variable and the destination of e ($e.destination$). We define *slack*(e) by:

$$slack(e) = e.source.distance - e.destination.distance + e.weight$$

Any cycle $C = [v_1, v_2, \dots, v_n]$ in G' will have its length equal to zero. To see that, let us first make some definitions:

- $v_1..v_n$ are vertices
- For every i , we have v_i is connected to v_{i+1} and v_n is connected to v_1
- $w(v_i, v_{i+1})$ is the weight of the edge between v_i and v_{i+1} . It is a shortcut for $Edge(v_i, v_{i+1}).weight$
- $d(v_i)$ is the length of the shortest path between a vertex (the *artificial* one) and v_i . It is a shortcut for $v_i.distance$

The length of a cycle C is the sum of the edge weights in the cycle:

$$Length(C) = w(v_1, v_2) + w(v_2, v_3) + \dots + w(v_n, v_1);$$

```

input : G=(V,E): Graph
output: s: Set of Equalities
data  : G': Graph
data  : E': Edge
data  : eq: Equality
data  : SCCs: Set of SCC

// Mount graph G' from E' and its vertices
1 foreach Edge e in E do
2   | if e.source.distance - e.destination.distance + e.weight = 0
3   | | E'.Insert(e);
4   | end
5 end
6 G' := Graph(E');

// Look for equalities in each SCC
7 SCCs := G'.SCC();
8 foreach SCC scc in SCCs do
9   | // sort vertices of scc by distance
10  | Sort(scc.V, IncreasingDistance());
11  | for i = 1 to scc.V.Size()-1 do
12  | | if scc.vi.distance = scc.vi+1.distance then
13  | | | eq := Equality(scc.vi, scc.vi+1);
14  | | | if NotGeneratedYet(eq) then
15  | | | | s.Insert(eq);
16  | | | | eq.SetPremises(FindPremises(G', scc.vi,
17  | | | | | scc.vi+1));
18  | | | end
19  | | end
20 end
21 return s ;

```

Algorithm 7: GenerateEqualities

Adding the terms $+d(v_i)$ and $-d(v_i)$ will not change the result of the length, because they cancel each other. So, doing this we have:

$$\text{Length}(C) = d(v_1) - d(v_1) + d(v_2) - d(v_2) + \dots + d(v_{n-1}) - d(v_{n-1}) + w(v_1, v_2) + w(v_2, v_3) + \dots + w(v_n, v_1);$$

Now, we just rearrange the terms to see more precisely the next step and obtain:

$$\text{Length}(C) = [d(v_1) - d(v_2) + w(v_1, v_2)] + [d(v_2) - d(v_3) + w(v_2, v_3)] + \dots + [d(v_n) - d(v_1) + w(v_n, v_1)];$$

We can see that $[d(v_i) - d(v_{i+1}) + w(v_i, v_{i+1})]$ is exactly the *slack* defined before. We know that in G' , all the *slacks* are 0 because of the way G' was built. Therefore we have:

$$\text{Length}(C) = 0$$

The next step in the algorithm is to get the strongly connected components (SCCs). A graph G_r is strongly connected if for all pairs of vertices u and v from G_r there exists a path from u to v and also from v to u . We will call the SCCs of a graph G_r the maximal² strongly connected subgraphs of G_r .

Any two vertices in a SCC of G' will be in a cycle of length zero and the path between them is also the shortest path in the original graph G . That is the primary condition for two variables to be equal. They need to be in a cycle of length zero. Based on that, every pair of vertices in each SCC are potential candidates to be equal.

For checking if two variables v_1 and v_k are equal without having to calculate the shortest path between every pair of variables, we can use the *distance* information that we keep when looking for negative cycles. The shortest paths between them have to be zero, so:

$$\begin{aligned} 0 &= w(v_1, v_2) + w(v_2, v_3) + \dots + w(v_{k-1}, v_k); \\ 0 &= w(v_k, v_{k+1}) + \dots + w(v_{n-1}, v_n) + w(v_n, v_1); \end{aligned}$$

We know by the definition of *slack* that: $\text{slack}(v_i, v_{i+1}) = d(v_i) - d(v_{i+1}) + w(v_i, v_{i+1})$. Knowing that the *slacks* in G' are zero: $w(v_i, v_{i+1}) = -d(v_i) + d(v_{i+1})$. Thus, developing the previous expressions:

$$\begin{aligned} 0 &= -d(v_1) + d(v_2) - d(v_2) + \dots + d(v_{k-1}) - d(v_{k-1}) + d(v_k); \\ 0 &= -d(v_k) + d(v_{k+1}) - d(v_{k+1}) + \dots + d(v_n) - d(v_n) + d(v_1); \end{aligned}$$

Simplifying everything:

$$\begin{aligned} 0 &= -d(v_1) + d(v_k); \\ 0 &= -d(v_k) + d(v_1); \end{aligned}$$

Therefore, in any SCC of G' , two variables v_1 and v_k are equal if they have the same distance value, i.e., $d(v_1) = d(v_k)$.

Summarizing, Algorithm 7 will look in each SCC of G' for every pair of vertices. If they have the same distance, an equality is found. If this equality is new, it is added to the set of new equalities generated to be returned. Additionally, the set of premises is associated to it.

²Maximal here means that if we include any other vertex from the graph to the SCC, it will no longer be a strongly connected component.

Strongly Connected Component Algorithm

Algorithm 8 shows a pseudo-code for finding the strongly connected components (SCCs) of a graph. It is a classical algorithm and its complexity is linear in the number of edges [67].

```

input :  $G=(V,E)$ : Graph
output:  $s$ : Set of SCC
data  :  $G^T$ : Graph
data  :  $E^T$ : Edge

// Mount  $G^T$  by creating new edges changing the source
// and destination of  $E$ 
1 foreach Edge  $e$  in  $E$  do
2 |  $E^T$ .Insert(Edge( $e$ .destination,  $e$ .source));
3 end
4  $G^T :=$  Graph( $V, E^T$ );

// First DFS
5 ResetVisitedFlags( $V$ );
6 foreach Vertex  $v$  in  $V$  do
7 | if  $v$ .visited = false then
8 | |  $v$ .visited := true ;
9 | | DFS( $G, v$ );
10 | end
11 end

// Second DFS, by decreasing finish time of the first
// DFS
12 Sort( $V, DecreaseTime()$ );
13 ResetVisitedFlags( $V$ );
14 foreach Vertex  $v$  in the sorted  $V$  do
15 | if  $v$ .visited = false then
16 | |  $v$ .visited := true ;
16 | | // Do a DFS and creates a SCC from the vertices
16 | | visited in this iteration
17 | |  $s$ .Insert(DFS( $G^T, v$ ));
18 | end
19 end
20 return  $s$ ;

```

Algorithm 8: FindStronglyConnectedComponents

It runs a series of depth first searches (DFS) using a graph G starting from each vertex that has not been visited yet. Later, another series of DFS is done, but now using the transpose set of edges E^T and choosing the vertex in decreasing order of “finished visit” time of the first DFS. In this second

series of DFS, all the vertices reachable on each DFS belong to the same SCC.

Depth-First Search (DFS) Algorithm

Algorithm 9 shows how a depth-first search (DFS) works. The DFS is a search algorithm that explores each branch as deeply as possible before backtracking, see, e.g., [19]. Starting in a vertex v , it will mark as visited every vertex reachable from v . Additionally, when a vertex u cannot continue the search, i.e., all its neighbors were already visited, u will have a “finished visit” time set.

```

input :  $G=(V,E)$ : Graph
input :  $v$ : Vertex
output:  $s$ : Set of Vertex

1 foreach OutgoingEdge  $e$  in  $V$  do
2   | if  $e.destination.visited = false$  then
3   |   |  $e.destination.visited := true$  ;
4   |   |  $s = e.destination + DFS(G, e.destination)$ ;
5   |   | end
6 end
7  $v.timeFinished := NextTime()$ ;
   // Return the set of visited vertices reachable from  $v$ 

8 return  $s$  ;

```

Algorithm 9: DFS

Explanation of an equality

Given an equality between two variables, the premises of this equality are the constraints that generated it. Two variables u and v are equal if the length of the shortest paths from u to v and from v to u are 0.

Then, for constructing the set of premises, it is only necessary to extract the constraints related to each edge in the shortest path between u and v and between v and u . Algorithm 10 shows a pseudo-code for this.

It does two breadth-first searches (BFS). First starting from u and second starting from v . That will save in G the predecessors corresponding to (one of) the shortest paths between u and v and between v and u . Later, this information is used to go through the paths and collect the constraints associated to each edge in the paths.

An equality can have more than one minimal set of premises (see Figure 6.12 for an example). Using BFS to find the shortest path (number of edges) will allow finding the *minimum* set of premises.

```

input :  $G=(V,E)$ : Graph
input :  $u, v$ : Vertex
output:  $s$ : Set of Constraints

// Update the predecessor
1 BFS( $G, u, v$ );
2 BFS( $G, v, u$ );

// Get the set of premises
3  $s := \text{GetConstraints}(G, u, v) + \text{GetConstraints}(G, v, u)$ ;
4 return  $s$  ;

```

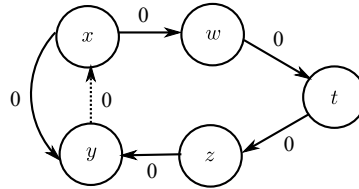
Algorithm 10: FindPremises

Figure 6.12: In this graph, representing a set of constraints, an equality can be found after the edge $(y, x, 0)$ is added. Two sets of premises can be returned as a proof for $x = y$ (there are two shortest paths with length 0): $\{x - y \leq 0, y - x \leq 0\}$ and $\{x - y \leq 0, w - x \leq 0, t - w \leq 0, z - t \leq 0, y - z \leq 0\}$.

The BFS is a search algorithm that can be used to find the shortest path (related to the shortest number of edges), see, e.g., [19]. It executes the search by visiting all neighbors first and then visiting the neighbors of the neighbors and so on, until it finds the goal. For this, it uses a queue (FIFO - First In First Out). Algorithm 11 shows a pseudo-code. The goal is to find the destination vertex.

Algorithm 12 shows a pseudo-code for getting the constraints in a path. It is similar to the Algorithm 6 that gets the conflict set. It just follows the path using the predecessor information that was previously recorded and collect the constraints from the edges in the path.

6.2.5 Model-Equality Generation

In the previous section we saw how to generate equalities for difference logic. If the domain of the variables is the real numbers, then generating only equalities is enough for a decision procedure to be complete in a Nelson and Oppen framework. But if we are working with integers, that is not enough. We saw in Section 4.2 how model-equalities can be used to achieve completeness and all the process involving them. In the current section

```

input : G=(V,E): Graph
input : source, destination: Vertex
data  : q: Queue of Vertex
data  : u: Vertex

1 source.predecessor := NULL ;
2 q.Add (source);
3 while q.Empty() = false do
4   | u := q.Remove();
5   | foreach OutgoingEdge e in u do
6     | if e.destination.visited = false then
7       | e.destination.visited := true ;
8       | e.destination.predecessor := u ;
9       | if e.destination = destination then
10      | | return ;
11      | end
12      | q.Add(e.destination);
13    | end
14  | end
15 end

```

Algorithm 11: BFS

```

input : G=(V,E): Graph
input : source, destination: Vertex
output: s: Set of Constraints
data  : pred: Vertex

1 while source  $\neq$  destination do
2   | pred := destination.predecessor ;
3   | s.Insert (Edge(pred, destination).constraints);
4   | destination := pred ;
5 end
6 return s ;

```

Algorithm 12: GetConstraints

we focus on aspects for difference logic. We show how to generate model-equalities for difference logic making use of the same data structures used in the satisfiability check.

We start by taking a look at an example. Given the following set of constraints $\{d - a \leq 1, c - a \leq -1, b - a \leq -1, e - b \leq -1, e - c \leq -2, f - e \leq 4\}$, we build a graph from it, running the satisfiability check algorithm to obtain the distance information. Figure 6.13 shows the graph, with the distance information in the squares.

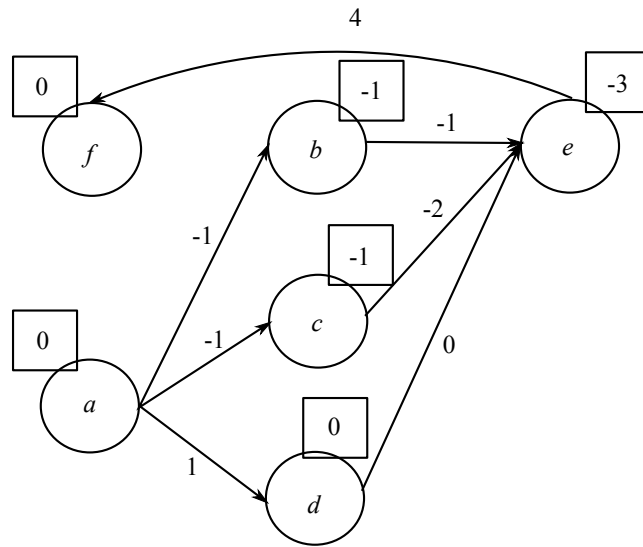


Figure 6.13: Graph from $\{d-a \leq 1, c-a \leq -1, b-a \leq -1, e-b \leq -1, e-c \leq -2, f-e \leq 4\}$. No negative cycle detected.

The simplest way to build a consistent model out of this situation is to use the current distance information from the data structure. There is no extra cost for building the model, and we can simply generate the model-equalities out of the variables that have the same value. From this example we have the model shown in Table 6.2, from which we can generate the model-equalities $a = d = f$ and $b = c$. If the decision procedure is only working with constraints of the kind $\{=, \leq, \geq, <, >\}$, all the constraints will be represented in the graph and the model will be always consistent.

Model	
Variable	Value
a	0
b	-1
c	-1
d	0
e	-3
f	0

Table 6.2: Arithmetic model from example of Figure 6.13.

Once generated, the model-equalities are propagated and if no other decision procedure disagrees with the model, the process finishes. Otherwise, the model-equalities will be removed and the model modified. A model is

wrong when one variable is equal to another when it should not be. The decision procedure will handle this demand to modify the model by accepting constraints of the kind $\{\neq\}$, disequalities. Disequalities cannot be incorporated to the graph, so they will be treated separately.

The easiest way to handle disequalities in our case is, after the model is built, when we are looking at the values of the variables to generate the model-equalities, we also look at the disequalities. If two variables have the same value, but there is a disequality between them, then the decision procedure generates a lemma that will later allow itself to indirectly incorporate the disequality in the graph, and therefore, modify the conflicting model.

A disequality $x \neq y$ can be arithmetically understood as a disjunction of inequalities $x < y \vee x > y$. Handling disjunctions in the decision procedure would be complicated, but we know that a SAT-solver can handle them very easily and efficiently. Also, by delegating this task to the SAT-solver we automatically make use of efficient techniques such as learning, backtracking, etc. So, for handling a disequality $x \neq y$, we generate a lemma of the kind $x \neq y \implies x < y \vee x > y$. By doing this, we know that in future interactions with the SAT-solver, the difference logic decision procedure will receive either $x < y$ or $x > y$, and with this extra information, that can be incorporated to the graph, it will correct the model, fixing the conflict that there was with the disequality. This lazy process of creating lemmas to handle split cases that could not be easily done by the decision procedures is called *splitting on demand*, see [4].

What was described so far is how it is implemented in the veriT solver. We have no cost for building the model and we can extract the model-equalities in linear time. We can see a pseudo-code for this process in Algorithms 13 and 14.

6.2.6 Generating fewer model-equalities

With some extra processing we can generate better models. We call a model better if from it we have fewer model-equalities. Having fewer model-equalities is better because it reduces the probability of having some decision procedure disagreeing with one of model-equalities, potentially reducing the time lost correcting the models. The ideas presented here are not yet implemented but we think they will significantly improve the solver.

For having better models we need to change the values of the variables. They will no longer necessarily be equal to the distance. The first observation we can make is that if there is nothing setting an upper bound to a variable, then we can set the value of this variable as high as wanted. A variable x has no upper bound limit if for all the constraints³ in the decision procedure there is no one of the kind $x - v \leq c$, for any variable v or

³Considering that all the constraints were translated to constraints of the kind \leq .

```

input : G =(V,E): Graph
input : D: Set of Disequalities
output: s: Set of ModelEqualities
data : values: Multimap of < Value, Variable >
data : v1, v2: Variable
data : Meq: ModelEquality

1 foreach Variable var in V do
2 | values.Add(var.value, var);
3 end
4 foreach Value val in values do
5 | for i = 1 to values.Count(val)-1 do
6 | | v1 := values.ElementOfAt(val,i);
7 | | v2 := values.ElementOfAt(val,i +1);
8 | | Meq := ModelEquality(v1, v2);
9 | | if NotGeneratedYet(Meq) then
10 | | | s.Insert (Meq);
11 | | end
12 | end
13 end
14 if VerifyModel(D, s) = CONFLICTING_WITH_DISEQUALITIES
then
| // The model needs to be corrected
15 | return s := NULL;
16 end
17 return s ;

```

Algorithm 13: GenerateModelEqualities

constant c . And we can still extend this observation, if a variable y has its upper bound limited⁴ only by other variables with no upper bound, then we can state that y has no upper bound limit either.

A similar analysis can be done for the lower bound. If there is nothing setting a lower bound to a variable, then we can set the value of this variable as low as necessary. A variable x has no lower bound limit if for all the constraints in the decision procedure there is no one of the kind $v - x \leq c$, for any variable v or constant c . Extending the observation, if a variable y has its lower bound limited⁵ only by other variables with no lower bound, then we can state that y has no lower bound limit either.

In practice that means that any variable that has no upper or lower

⁴A variable y has its upper bound limited by variable v if there is a constraint $y - v \leq c$ for some constant c .

⁵A variable y has its lower bound limited by variable v if there is a constraints $v - y \leq c$ for some constant c .


```

input : D: Set of Disequalities
input : ME: Set of ModelEqualities
output: status: Status

1 status := NO_CONFLICT ;
2 foreach Disequality d in D do
3   | if d.v1.value = d.v2.value ∈ ME then
4     | // Generate a lemma to correct the model
5     | GenerateLemma(d.v1 ≠ d.v2 ⇒ d.v1 < d.v2 ∨ d.v1 >
6     | d.v2);
7     | status := CONFLICTING_WITH_DISEQUALITIES ;
8   | end
9 end
10 return status ;

```

Algorithm 14: VerifyModel

bounds can have its value set to any number. Or, in other words, we do not ever need to build model-equalities involving variables that have no upper or lower bound limits.

We can also translate this first observation to the graph representation we use for difference logic. A variable has no upper bound if the vertex representing it has no incoming edges. Vertices with incoming edges that comes only from other vertices with no upper bound limit have also no upper bound limit. Similar can be done to the lower bound case. In the end, if we remove all the vertices that have no upper or lower bound limit, we will have only vertices that belongs to strongly connected components having at least two vertices.

Following the reasoning, we can improve the difference logic model by running the strongly connected components algorithm and ignoring the strongly connected components that only have one vertex. Or, if one prefers, set these vertices to some arbitrarily different values. We can see how we can reduce the number of variables that we need to worry about when generating model-equalities in Figures 6.14 and 6.15. Figure 6.14 is the initial graph representation of an arbitrary problem and Figure 6.15 shows the reduced graph after running the strongly connected components algorithm and ignoring the components with size one.

The graph formed from the strongly connected components⁶ is acyclic, Figure 6.15 is an example. We can safely define a topological order⁷ for the components, like shown in Figure 6.16.

The values of the variables in the same strongly connected component

⁶Each strongly connected component is interpreted as one vertex

⁷Topological order is a linear ordering of its vertices in which each vertex comes before all vertices to which it has outgoing edges.

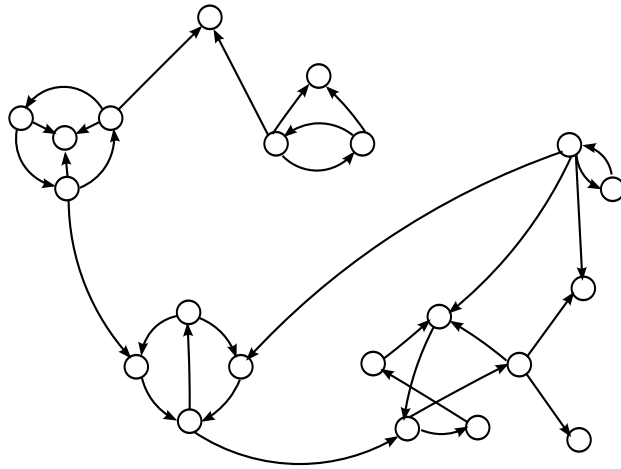


Figure 6.14: Graph representation of an arbitrary problem.

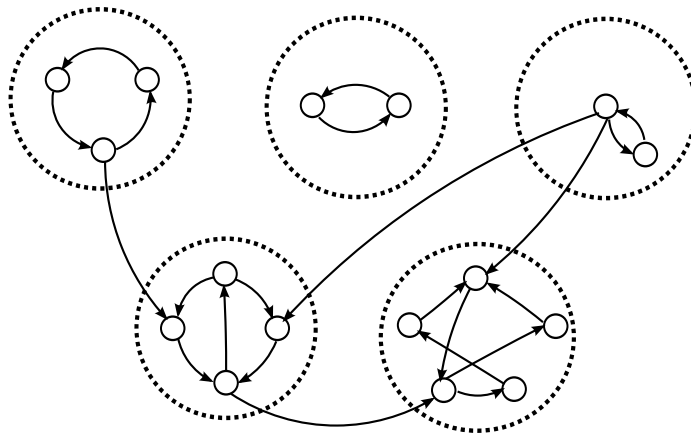


Figure 6.15: Graph from Figure 6.14 after removing the strongly connected components with one element. Each dashed circle represents a different strongly connected component.

depend on each other, but we can define a range of values where they can all fit. And since no variable is limited by a number⁸ we can set this range in any interval between $-\infty$ and ∞ . This can also be seen in Figure 6.16.

The last observation is that the strongly connected components may only limit each other by the upper or lower bound, but not both. Therefore, we can use one topological ordering and shift the range of the components as

⁸Difference logic constraints have always two variables, there are no constraints of the kind $x \leq c$, where c is a constant.

we want, so that we have no overlap. Now that we are sure that variables of different strongly connected components are set to values in different ranges, the decision procedure may limit itself to look for model-equalities only between variables of the same strongly connected component.

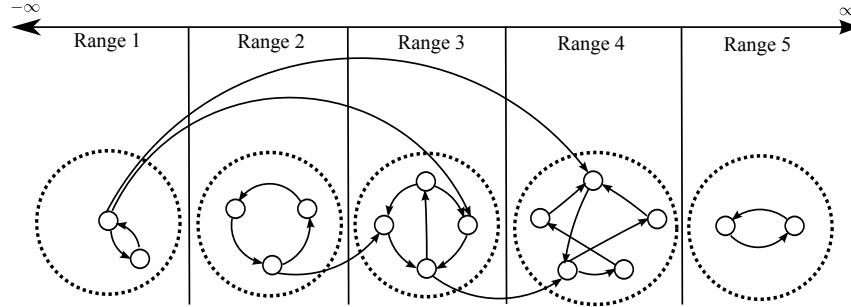


Figure 6.16: Graph from Figure 6.15 after defining a topological ordering. Each strongly connected component has a different range of values defined.

In practice, we do not need to worry about finding the ranges. We run the strongly connected components algorithm and use the distance information to look for model-equalities only between variables in the same component. One difference to Algorithm 7, for generating equalities, is that the one for equalities run over a modified graph, while the one for model-equalities runs over the original graph. We can see a pseudo-code in Algorithm 15.

We have shown an algorithm for finding a better model. In practice we did not modify the values of the variables. One can still try to modify the values of the variables directly in the decision procedure for trying to reduce even further the number of generated model-equalities, but should be careful to not invalidate the model, and never skip a valid model before affirming the problem is unsatisfiable. Future work is required to give better theoretical foundation and to evaluate experimentally the benefit of this method.

6.2.7 Theory propagation

One of the good new features decision procedures may have in SMT-solvers is theory propagation. If used wisely and the SAT-solver is well integrated with the decision procedures, theory propagation may improve considerably the efficiency of an SMT-solver. However, the operation to find implied literals should not be costly.

Using the graph theory presented in this chapter, it is easy to know if a difference logic literal is implied by a set of constraints. Let $x - y \leq c$ be the literal that we want to verify. If in our graph there is a path that goes from y to x with length c' , where $c' \leq c$, then $x - y \leq c$ is implied by the current set of constraints. This can be done by using a single-source shortest path

```

input : G=(V,E): Graph
input : D: Set of Disequalities
output: s: Set of ModelEqualities
data : values: Multimap of < Value, Variable >
data : v1, v2: Variable
data : Meq: ModelEquality
data : SCCs: Set of SCC

1 SCCs := G.SCC();
2 foreach SCC scc in SCCs do
3   values := NULL ;
4   foreach Variable var in scc.V do
5     | values.Add(var.value, var);
6   end
7   foreach Value val in values do
8     | for i = 1 to values.Count(val) - 1 do
9       | v1 := values.ElementOfAt(val,i);
10      | v2 := values.ElementOfAt(val,i + 1);
11      | Meq := ModelEquality(v1, v2);
12      | if NotGeneratedYet(Meq) then
13        | s.Insert (Meq);
14      | end
15    | end
16  | end
17 end
18 if VerifyModel(D, s) = CONFLICTING_WITH_DISEQUALITIES
19 then
20   | // The model needs to be corrected
21   | return s := NULL;
22 end
23 return s ;

```

Algorithm 15: GenerateModelEqualitiesImproved

algorithm, such as Dijkstra algorithm [29, 19], which can be implemented in $O(|E| + |V| \log |V|)$.

If there are many literals that can be implied, the single-source shortest path algorithm is going to be used many times. In this case, maybe it is better to use an all-pairs shortest path algorithm, such as Floyd-Warshall algorithm [71, 19], which can be implemented in $O(|V|^3)$.

However, the smartest strategy would be to adapt the incremental satisfiability check algorithm. The ideal would be to make it also produce, with little or no extra cost, the information necessary to perform theory propaga-

tion. The Floyd-Warshall algorithm is a natural example. Although slower, Floyd-Warshall algorithm can also be used to detect negative cycles and therefore unsatisfiability. Using it to check satisfiability would simplify the later check for implied literals significantly.

Although theory propagation has been used in a few solvers for a few years, it has just been implemented in our solver `veriT`, for the decision procedure of uninterpreted functions. The next step would be to implement it also for difference logic. The research direction would be to adapt the incremental satisfiability check algorithm presented in this chapter to simplify the later use of theory propagation.

6.3 Conclusion

Additionally to satisfiability check, there are many extra requirements to build a decision procedure for an SMT-solver. We have seen in this chapter all the details necessary to build a complete decision procedure for real and integer difference logic, fulfilling all the requirements.

It was presented an original⁹ incremental algorithm for the satisfiability check which produce enough information to simplify the task of many of the extra requirements. This chapter also contributes with precise details of how to generate model-equalities for difference logic and obtain a decision procedure that can be used in a combination framework, producing complete results.

The next chapter also presents the elements necessary to build a decision procedure. This time, the linear arithmetic theory is the focus. We will show how to build a decision procedure based on the simplex algorithm.

⁹It was designed in a period pre-thesis.

Chapter 7

Deciding linear arithmetic

We have seen previously, the elements necessary to build a decision procedure for the arithmetic fragment of difference logic. To go to the linear arithmetic domain and obtain more expressiveness, the algorithms change and are completely different from the ones of the difference logic theory. They are naturally more elaborate and less efficient, but we are still able to solve important size problems using decision procedures for linear arithmetic.

With linear arithmetic we gain more expressiveness. In this fragment we are allowed to use addition, subtraction and multiplication with no restriction, except for multiplication between variables. Another advantage is to use as many variables per constraint as necessary, instead of only two of difference logic.

We no longer work with graph theory as in difference logic. There are two main families of algorithms for linear arithmetic that have been used in current SMT-solvers, one based on the *Fourier-Motzkin elimination method* [22] and one based on the *simplex method* [20]. In this chapter, we take the direction of the simplex that has been shown in practice to be the most efficient method for linear arithmetic decision procedures lately [30, 61, 28, 51, 2]. We are going to show an original variation of the simplex designed and implemented in our SMT-solver `veriT`.

7.1 Introduction to the simplex method

The simplex method was created by George Dantzig in 1947 and is one of the most popular methods for linear programming. The name of the algorithm comes from the concept of simplex, that in geometry is a generalization of the notion of a triangle or tetrahedron to arbitrary dimension (Figure 7.1 shows a tetrahedron).

Typically, the simplex method solves linear programming problems, where we are supposed to maximize (or minimize) a function, restricted by some

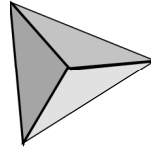


Figure 7.1: A tetrahedron or 3-simplex.

inequality constraints. An example of a linear programming problem is:

$$\begin{aligned}
 &\text{Maximize: } F = 3x + 2y + z \\
 &\text{Subject to: } 2x + y \leq 18 \\
 &\quad \quad \quad 2x + 3y \leq 42 \\
 &\quad \quad \quad 3x + y + 3z \leq 24 \\
 &\quad \quad \quad x, y, z \geq 0 \\
 &\quad \quad \quad x, y, z \in \mathbb{Q}
 \end{aligned}$$

From a geometrical point of view, each inequality makes a cut in the hyperspace. In the intersections, we have the edges and vertices that will form our geometrical convex object, a n -dimensional polytope, where n is the number of variables. The simplex algorithm starts at a vertex in the feasible region (the region delimited by the simplex) and walks along the edges of simplex, moving to vertices with higher objective functions. When it reaches the local maximum, by convexity it is also the global maximum, so the algorithm stops. Figure 7.2 shows an example in three dimensions (three variables). It shows how following the edges of a simplex we can find the global maximum of a linear function F .

There are only two exceptions when looking for optimal solutions: problems with multiple optimal solutions and problems with no optimal solution. Problems have multiple optimal solutions when the maximum of a problem lies on an edge, instead of a single vertex. Problems have no optimal solution when there is no maximum because some variables are unbounded and may have any arbitrarily large values. In this case, the “simplex” object representing the problem has infinite size.

Since the first version, multiple variations of the simplex method appeared, see e.g. [21, 47, 6, 60]. They were developed to reduce some limitations with the constraint and variable inputs, to make a wider range of problems solvable, to increase performance with smarter decision heuristics, to increase numerical accuracy, etc. The implementations for SMT-solvers are also based on different simplex versions. We cite as examples versions

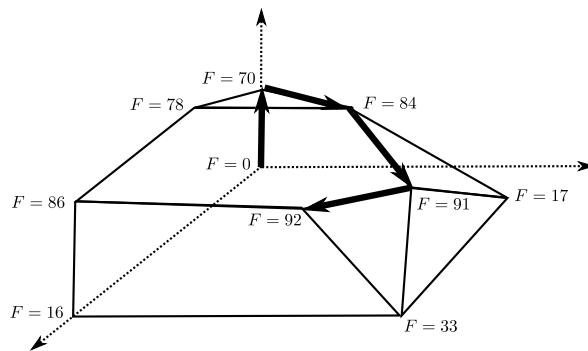


Figure 7.2: Starting at the origin, in the feasible region, the maximum of F is reached by following the vertices with higher function value.

based on dual simplex [30], primal simplex [61] and even using commercial floating point tools [11, 33, 46].

Following, the basic primal simplex is presented. We show how the method works before explaining the variation of the simplex we developed for our SMT-solver veriT.

7.2 The primal simplex

Consider the linear programming problem with two variables and three constraints. We want to maximize the Z function (i.e, get the optimal solution), subject to three constraints. Additionally, the simplex method imposes¹ the restriction on the bound of the variables, $x, y \geq 0$.

$$\begin{aligned} \text{Maximize: } & Z = 2x + 3y \\ \text{Subject to: } & -x + y \leq 5 \\ & x + 3y \leq 35 \\ & x \leq 20 \\ & x, y \geq 0 \end{aligned}$$

The first step of the algorithm is to transform the problem to a system of linear equations. We can do this by creating slack variables, changing inequalities like $x + y \leq 0$ to $x + y + s_1 = 0$, where $s_1 \geq 0$. The simplex method also imposes that all the new slack variables must be greater or equal to zero. After the transformations, the system obtained is:

¹Doing otherwise would imply in important modifications in the operations of the primal simplex method.

$$\begin{aligned}
&\text{Maximize: } Z = 2x + 3y \\
&\text{Subject to: } -x + y + s_1 = 5 \\
&\quad \quad \quad x + 3y + s_2 = 35 \\
&\quad \quad \quad x + s_3 = 20 \\
&\quad \quad \quad x, y, s_1, s_2, s_3 \geq 0
\end{aligned}$$

In the second step, a tableau is built from the system of equations. The tableau is in matrix form and represents the system of equations, see Figure 7.3. It provides a cleaner view of the problem and we will use it to perform the algorithms operations. The restriction $x, y, s_1, s_2, s_3 \geq 0$ is omitted.

	Z	x	y	s_1	s_2	s_3	
s_1	0	-1	1	1	0	0	5
s_2	0	1	3	0	1	0	35
s_3	0	1	0	0	0	1	20
Z	1	-2	-3	0	0	0	0

Figure 7.3: Initial tableau created from the translation of the system of linear equations.

At this point we can define two sets for classifying the variables that we will constantly refer to in the simplex operations: the basic variables and the non-basic variables. Every equation in the tableau has exactly one basic variable and the basic variable of an equation e only appears in e . The non-basic variables have no such restriction. The idea is that the basic variables are defined by a linear combination of non-basic variables.

We say that a variable is in the basis if it is a basic variable. Non-basic variables may enter the basis making a basic variable to leave the basis when performing pivot operations that will be explained later. In the tableau, the variables in the basis are shown in the first column. They may also be identifiable by checking the columns for variables that only appear in one of the constraints (in one of the rows there is a 1 and in the remaining there are only 0). The construction of the tableau makes it direct to set the variables Z and s_i as the initial basic variables. Figure 7.4 highlights this.

At any point, we have a basic solution associated to the tableau. In a basic solution, all non-basic variables are assigned to the value zero, and the value of the basic variables can be obtained by dividing the value in the rightmost column by the non-zero value in the variable column, in our case always 1. The basic solution of the initial tableau can be seen in Figure 7.5.

	Z	x	y	s_1	s_2	s_3	
s_1	0	-1	1	1	0	0	5
s_2	0	1	3	0	1	0	35
s_3	0	1	0	0	0	1	20
Z	1	-2	-3	0	0	0	0

Figure 7.4: Identifying basic variables in the tableau.

	Z	x	y	s_1	s_2	s_3	
s_1	0	-1	1	1	0	0	5
s_2	0	1	3	0	1	0	35
s_3	0	1	0	0	0	1	20
Z	1	-2	-3	0	0	0	0

$x, y = 0$
 $\implies s_1 = 5/1 = 5$
 $\implies s_2 = 35/1 = 35$
 $\implies s_3 = 20/1 = 20$
 $\implies Z = 0/1 = 0$

Figure 7.5: Basic solution of the initial tableau, in Figure 7.3

Third step is to select the variable that will enter the basis.

We look at the numbers in the bottom row, the row of the maximization function, and look for the most negative number. See Figure 7.6. If there are no negative numbers in the bottom row, it means the optimal solution was reached and the algorithm stops.

To see clearly the reason, rewrite the last row back to the equation form. We obtain $Z - 2x - 3y = 0$, or, $Z = 2x + 3y$. Notice that to maximize the value of Z , we need to increase the value of x or y , that are the variables with negative numbers in the last row of the tableau. Variables with positive numbers would need to be decreased. But as they are non-basic variables (have value zero) and additionally have lower bound equal to zero, they actually cannot be decreased. Therefore if there are no negative numbers in the bottom row, the maximization function cannot be improved and the algorithm stops.

The most negative number is related to the variable that has the potential to increase the most the maximization function in a single step. It is not necessarily the path that will lead to the optimal solution the fastest, but it is a general good greedy heuristic defined in the original simplex method.

The fourth step is to select the variable that will leave the basis. We perform a test to determine which variable is limiting the most the change of value of the entering variable, as we do not want to violate the constraints. The test is as follows: we calculate ratios by dividing the positive numbers in the column of the entering variable by the number in the same row in the rightmost column. The variable with the lowest ratio is the variable restricting the most the change of value of the entering variable and will leave the basis, see Figure 7.7. If there is no positive entry the

	Z	x	y	s_1	s_2	s_3	
s_1	0	-1	1	1	0	0	5
s_2	0	1	3	0	1	0	35
s_3	0	1	0	0	0	1	20
Z	1	-2	-3	0	0	0	0



 Candidate variables
to enter the basis


Figure 7.6: Selecting the variable to enter the basis. y will be the variable choose to enter.

solution is unbounded and the algorithm stops.

Remember that non-basic variables have value zero and basic variables have their values calculated from the equations. The ratio test is simply the calculation of the new value of the entering variable when we reduce the value of the leaving variable to its minimum, i.e., zero. Therefore, the lowest ratio is the maximum non violating value that the entering variable may have. In the original simplex method, it is required to always stay in the feasible region while searching for the optimal solution. Violating a constraint means we leave the feasible region, or in a geometrical point of view, we no longer are at a vertex of the simplex that represents the problem.

We do not use zero numbers in the ratio test because dividing by zero means that the entering variable has no restriction and could have infinite value. That is why if there is no positive entry in the ratio test the solution is unbounded and the algorithm stops.

	Z	x	y	s_1	s_2	s_3	
s_1	0	-1	1	1	0	0	5
s_2	0	1	3	0	1	0	35
s_3	0	1	0	0	0	1	20
Z	1	-2	-3	0	0	0	0


 Entering variable

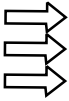
Ratio test

 5/1
 35/3
 --

Figure 7.7: Selecting the variable to leave the basis: calculating the ratio. s_1 will be the variable chosen to leave.

The fifth step is to pivot. We do linear arithmetic combinations to express the equations of the tableau in terms of the new set of basic variables. The resulting tableau is shown in Figure 7.8. Variable y entered

the basis and variable s_1 left the basis. The equations were rewritten so y only appears in one constraint. The current solution has the value of the maximization function Z 15. However, there is still room for improvement as there is a negative number in the bottom row. **We repeat steps 3-5 until no negative numbers are found in the bottom row.**

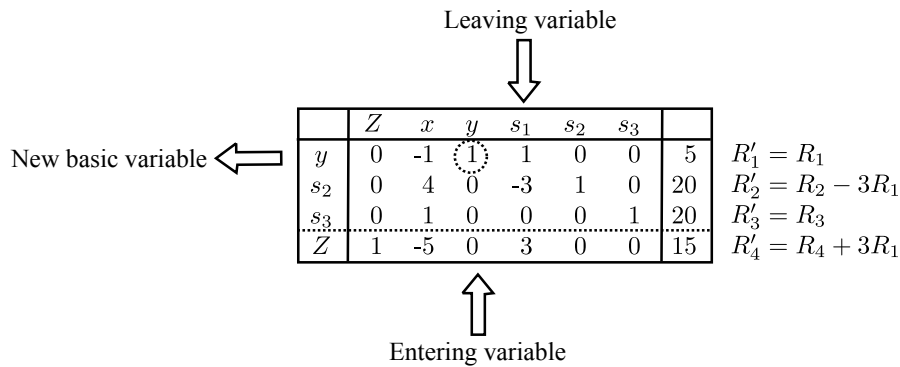


Figure 7.8: The resulting tableau after doing the pivot step. The basic variable s_1 was replaced by y and the equations had linear combinations performed to remove y from them.

Repeating the steps 3-5 twice, we get the tableaux shown in Figure 7.9 and Figure 7.10.

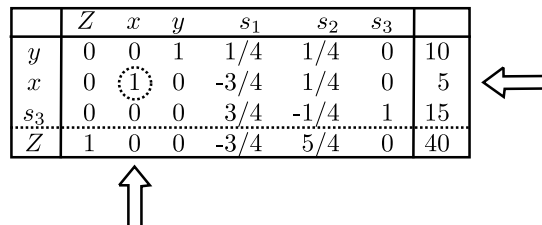


Figure 7.9: The resulting tableau after replacing the basic variable s_2 by the non-basic variable x .

At this point, there are no longer negative variables in the bottom row, so the algorithm stops. The solution associated to the last tableau shows us that the optimal solution Z is 55 and that happens when variable x has value 20 and variable y has value 5.

We can also see what is happening geometrically when running the simplex method on this example. Fixing as the axis the variables x and y , we can have a 2-dimensional representation of this same example in a Cartesian plane. The initial state can be seen in Figure 7.11.

	Z	x	y	s_1	s_2	s_3	
y	0	0	1	0	$1/3$	$-1/3$	5
x	0	1	0	0	0	1	20
s_1	0	0	0	1	$-1/3$	$4/3$	20
Z	1	0	0	0	1	1	55

Figure 7.10: The resulting tableau after replacing the basic variable s_3 by the non-basic variable s_1 .

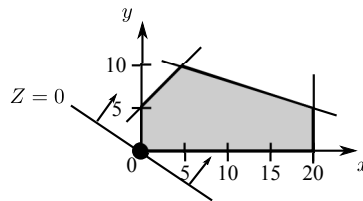


Figure 7.11: Geometric representation of the first tableau, the one of Figure 7.3.

The highlighted area, delimited by the constraints of the problem, is the feasible region. The start vertex is $(x = 0, y = 0)$ and we move toward other vertices whenever we can maximize the objective function Z . The maximization function can also be seen in the figure and its arrows point to the direction where Z can be maximized.

The result of the first pivot can be seen in Figure 7.12. The variable y is now a basic variable and has a value different from zero. The objective function Z is now 15.

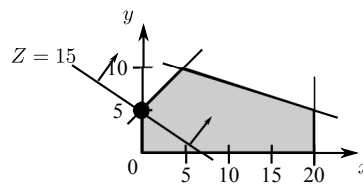


Figure 7.12: Geometric representation after the first pivot operation. Same state of the tableau of the Figure 7.8.

The next pivot result in the the graph of Figure 7.13. The variable x enters the basis and now has value 5. The value of variable y changes to 10 and Z goes to 40.

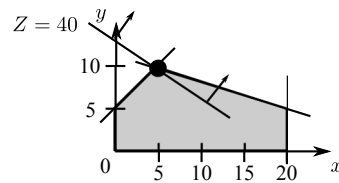


Figure 7.13: Geometric representation after the second pivot operation. Same state of the tableau of the Figure 7.9.

The result of the last pivot can be seen in Figure 7.14. We reach the vertex where Z has its maximum possible value when obeying the constraints. The graph represents perfectly the final solution, where $Z = 55$ with $x = 20$ and $y = 5$.

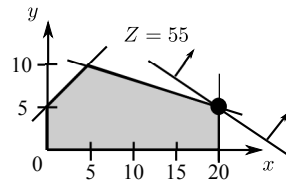


Figure 7.14: Geometric representation of the final state, after the last pivot operation. Same state of the tableau of the Figure 7.10.

This geometric representation can help understanding how the simplex method works. Notice however, that not all the information can be found in this graph representation. There is no information about the artificial slack variables (we could have included extra axis to represent more variables). The graph does not show which variables are in the basis. We know that a variable is in the basis if it has value different from zero (by definition of the primal simplex), but otherwise we cannot state the opposite.

One interesting observation comes from how the solution changes. Notice that the direction that the solution moves depends on the current basis. Every time there is a pivot, changing the basis, the solution can move in another direction, going from one vertex to another.

One thing that may happen during the simplex method is *degeneracy*. It happens when at some moment we have a basic variable with value zero. This may make the ratio test value zero and as result, the value of the maximization function after the pivot will not increase. Geometrically this happens when there are two or more vertices in the same point in the space, due to the intersection in the same point of three or more equations. Depending of which variables are chosen to enter and leave the basis, we may change of vertex but remain in the same coordinate point. Adding the con-

straint $y \leq 10$ to the previous problem, creates a degenerate point that we can see in Figure 7.15.

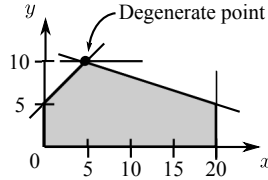


Figure 7.15: Degenerate point in the simplex.

Degeneracy happens with a certain frequency but it is not a serious issue unless it causes cycling, returning to a state that it has been before. But cycling does not happen often in practice. Nevertheless, there are methods for avoiding cycling, like Bland’s rule [13]. Bland’s rule works by giving an ordering to the variables and in case of ties in the minimum rate test, choosing the “smallest” variable to leave the basis.

In this section we have seen the primal simplex method and some details of interpretation. In the following section we are going to show how to create a decision procedure for linear arithmetic based on the simplex method but that has some important differences.

7.3 Incremental satisfiability check

In this section we show an original variation of the simplex algorithm. Our goal is to adapt the simplex method and to build a decision procedure for linear arithmetic. The first thing we need to adapt is the restrictions² about the constraints that are found in the primal simplex. Our procedure must: have variables initially unbounded (and not initially lower bounded by zero); and accept also equations and disequalities (and not only inequalities). The second important adaptation is to make the procedure incremental: as we have seen in Chapter 5, this is a fundamental requirement for efficiently integrating a decision procedure into an SMT-solver. In the following sections, we will also see how to extract important information that is also required like conflict set, model-equalities, etc.

Continuing with the differences, there is also an important relaxed restriction in comparison with the primal simplex: we no longer need to get an optimal solution. Any feasible solution is enough to detect the satisfiability. The simplex method works by improving the solution step-by-step, only stopping when the optimal solution is found. The relaxed restriction is

²Some of the restrictions can already be easily relaxed in some variations of the simplex or as pre-processing.

great because now the algorithm can stop even earlier, whenever a solution is satisfiable.

We explain the details of how it works and at the same time present an example. Through this section, we check the satisfiability of the following set of constraints.

$$\begin{aligned} -x + y &\leq 5 \\ x + 3y &\geq 35 \\ x &\geq 0 \\ y &\leq 5 \\ 3x + 3y &= 120 \end{aligned}$$

The algorithm incorporates one constraint at a time and if it finds a conflict, it stops immediately. For each new constraint, there are several steps that follow.

The first step is to verify if there are new variables. For each new variable, we initialize the lower and upper bounds to minus and plus infinity, respectively. We also initialize the current value of these new variables by setting them to zero. In the case of the first constraint $-x + y \leq 5$, we get two new variables x and y . To help to keep track of the value and bounds of the variables, we create a table that will be shown as the example continues. Its initial state is in Figure 7.16.

	Bounds		
var	lower	value	upper
x	$-\infty$	0	∞
y	$-\infty$	0	∞

Figure 7.16: Just after $-x + y \leq 5$ is added to the problem, the new variables x and y have value and bounds initialized.

We still have the idea of basic and non-basic variables. A basic variable only appears in one constraint while a non-basic variable may appear in several ones. Just created variables are always non-basic at the beginning.

The main difference to primal simplex is that a non-basic variable may have a value different from zero. We do not have the restriction of lower bound equals to zero anymore, so variables may have values decreased beyond zero. A basic variable continues to have its value calculated from the non-basic variables, but as the non-basic variables may have values different from zero, the calculation is not as direct as before.

The second step is to normalize the new constraint. We do it by replacing the basic variables by their equivalent expressions containing

only non-basic variables. As we are processing the first constraint and still do not have basic variables, the constraint $-x + y \leq 5$ is already normalized and therefore remains the same.

The third step is to set the new constraint as the objective function and try to satisfy it. Briefly saying, we try to satisfy the new objective function (which we may also call goal) by changing the values of the variables and doing pivot operations when necessary. It is not very different from the primal simplex, but due to a few changes we have incorporated, some details are not the same.

Back to the example, because of the current values of the variables, the goal is already satisfied. If we replace the values and evaluate the expression $-x + y \leq 5$, we obtain $-0 + 0 \leq 5$ which is true.

One of the good aspects of our incremental algorithm is that in many situations, like the previous one, the new constraint is already satisfied by the current problem, so no costly operation is necessary for a complete check of satisfiability. We will see more complex scenarios and details of step three when we process the remaining constraints.

The fourth step is to incorporate the new constraint to the current set of constraints. The way we do it will depend if it is an equation, inequality or disequation.

If the constraint is a disequation, we just save it apart to use it later. Disequations cannot be directly incorporated into the simplex, so we will only use them later for doing some final verifications.

If the constraint is an inequality, we create a slack variable to transform the inequality into an equation before incorporating it. Then, we set a value and a bound to the slack variable and choose the new slack variable as the basic variable of the constraint. The current value of the new slack variable is set to the difference between the evaluation of the variables and the constant term. In the case of our example, the variables of the expression $-x + y \leq 5$ evaluates to 0, so a new slack variable s_1 is set to 5 and the lower bound to 0, resulting in the new equation $-x + y + s_1 = 5$. Notice that if the inequality was $-x + y \geq 5$, the resulting new equation would be $x - y - s_1 = -5$, with $s_1 > 0$ and the value of s_1 set to 5.

If the constraint is already an equation, we do not need to create a slack variable. We just choose one of the variables to be the basic variable and normalize the previous constraints (like in step two) to remove the new basic variable from these constraints, if necessary.

There are two particular cases in this step. The first is if the constraint contains only one variable, we can adjust the bounds of the variable accordingly and then discard the constraint. The second is if the constraint contains no variable³, as it is a satisfying expression with no variable, we

³Additionally to user given constraints, this may occur because of operations like normalization and/or pivoting. The result can be an objective function containing no variable,

can simply discard it.

The result of processing the first constraint can be seen in Figure 7.17. It shows the constraint in the form of tableau, next to the table with the bounds and values of the variables.

	x	y	s_1	
s_1	-1	1	1	5

Bounds			
var	lower	value	upper
x	$-\infty$	0	∞
y	$-\infty$	0	∞
s_1	0	5	∞

Figure 7.17: The result of processing the constraint $-x + y \leq 5$.

At the same time, we can see geometrically what this processing means in a two dimensional graph shown in Figure 7.18. Simply adding the constraint does not make our current position ($x = 0, y = 0$) invalid (out of the feasible region). As we are not interested in finding the optimal solution, but just a solution, the current solution ($x = 0, y = 0$) does not need to change.

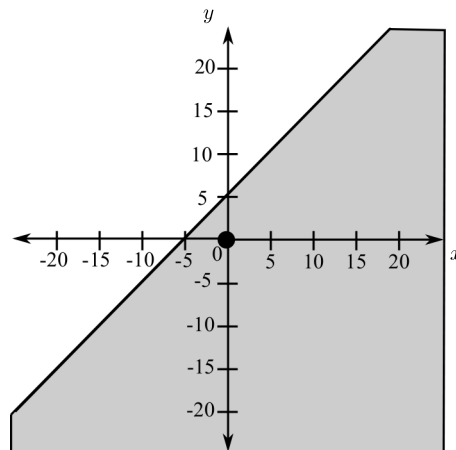


Figure 7.18: Geometric representation of the problem after adding the constraint $-x + y \leq 5$. The highlighted area represents the current feasible region and the point at the origin indicates the current value of user variables x and y .

The problem so far is satisfiable, so we continue processing the constraints. The second constraint is once again an inequality, $x + 3y \geq 35$.

In the step one, we find no new variable. In the step two, there are no basic variables in the constraint $x + 3y \geq 35$, so it is already normalized.

like e.g., the expression $0 = 0$.

We go then to step three. The constraint $x + 3y \geq 35$ is not currently satisfied by the values of the variables as it evaluates to $0 \geq 35$. To try to satisfy the new constraint, we will start a sequence of operations to change the values of the variables, while respecting the previous constraints.

We set $x + 3y \geq 35$ as our goal. That means we will try to maximize the left side of the expression until it evaluates to the same value in the right side. What we do now is similar to what we did in the primal simplex. We perform a sequence of sub-steps in a loop until the new constraint is satisfied or until we conclude there are no solution. The tableau reflecting the current state of the algorithm is shown in Figure 7.19.

	x	y	s_1	
s_1	-1	1	1	5
Max	1	3	0	35

	Bounds		
var	lower	value	upper
x	$-\infty$	0	∞
y	$-\infty$	0	∞
s_1	0	5	∞

Figure 7.19: The state of the tableau after the constraint $x + 3y \geq 35$ was added. We want to maximize the left side of the expression in the last row so that it reaches the same value of the right side.

Sub-step 3.1 is determining a variable to enter the basis. We look at the variables that are currently in the goal to determine which ones we can change the value so that we can get closer to the goal. **If none of the variables in the goal can change its value we determine the problem to be unsatisfiable.** This happens if all the variables in the goal are directly limited by their bounds. If variables cannot change their value to satisfy the new constraint, we can conclude the problem has no solution and therefore the algorithm stops. Otherwise, determining which variable can change its value is done as following.

First, we check the sign of the coefficients to know if we need to increase or decrease the value of each variable. In our example, both x and y have positive coefficients and as we want to maximize, we need to increase their values.

Ideally, we would pick the variable to enter the basis that can get us fastest to the goal. In practice, calculating this information precisely would be very costly, so we use a heuristic. In the primal simplex, we simply choose the variable with the greatest coefficient. But in our case, as we now may have upper limits to the variables, we do a bit differently.

We use the following heuristic test. For each variable in the goal, we multiply the coefficient by the difference between the bound limit and the current value of the variable. We pick the variable with the highest result from the test. In case of a tie, we choose the variable with greatest coefficient.

Figure 7.20 shows the calculation to get these values for both variables

x and y . As both x and y are unbounded, they end up having the same value. To break the tie, we choose y to enter the basis since it has a greater coefficient in the goal.

	x	y	s_1	
s_1	-1	1	1	5
Max	1	3	0	35

\Downarrow
 Entering variable
 \Downarrow
 test

Bounds			
var	lower	value	upper
x	$-\infty$	0	∞
y	$-\infty$	0	∞
s_1	0	5	∞

\Downarrow
 $3(\infty - 0) = \infty$
 \Downarrow
 $1(\infty - 0) = \infty$

Figure 7.20: Deciding which variable to enter the basis.

Sub-step 3.2 is determining a variable to leave the basis. The basic variable chosen to leave the basis will be the one that is limiting the most the change of value of the entering variable. The description of how to determine it follows.

We perform a test involving each constraint. The test is similar to the one in the primal simplex. The biggest difference is that now the calculation is done taking into consideration that the variables may be unbounded or have bounds different from zero. We describe here the test for the case where we want to increase⁴ the value of the entering variable.

For each constraint that contains the entering variable, we first check the signs of the coefficients in both entering and basic variable. We have to compensate every change in the value of the entering variable by changing also the value of the basic variable. In this way, we maintain the equations valid, keeping our solution inside the feasible region. If the signs are the same, the basic variable will also increase its value, and for the upcoming calculus, we take in consideration the upper bound. Otherwise, if signs are opposites, the basic variable will have its value decreased, and in this case we pay attention to the lower bound.

For the calculation, in each constraint, we multiply the coefficient of the basic variable c_{bv} by the difference between the value v_{bv} and the bound b_{bv} of the basic variable and divide the result by the coefficient of the entering variable c_{ev} in the constraint. This calculation informs what would be the value of the entering variable when replaced by each of the current basic variables. The basic variable that leads to the smallest result in this calculation will be the one chosen to leave the basis, as it is the variable limiting the most the change of value of the entering variable and, as it was explained before, we want to change the value of the entering variable, while respecting all the constraints, the maximum possible. The new value of the entering variable v'_{ev} will be the sum of the old value v_{ev} and the result of

⁴It is very easy to adapt it for the case we want to decrease the value of the variable.

this calculation done with the leaving variable.

$$\begin{aligned}
 c_{ev}(v_{ev} - v'_{ev}) &= -c_{bv}(v_{bv} - v'_{bv}) && \# \text{Balance in the change of values} \\
 c_{ev}(v_{ev} - v'_{ev}) &= -c_{bv}(v_{bv} - b_{bv}) && \# \text{The new value of the BV is its bound} \\
 v_{ev} - v'_{ev} &= -c_{bv}(v_{bv} - b_{bv})/c_{ev} \\
 v'_{ev} &= v_{ev} + (c_{bv}(v_{bv} - b_{bv})/c_{ev}) && \# \text{The new value of the entering variable}
 \end{aligned}$$

In the case of our example, we only have one constraint so far. We know that s_1 will leave the basis, but we do the test to obtain the new value of the entering and leaving variables. Figure 7.21 shows how the test is done.

		Leaving variable test		
		$\implies v'_y = 0 + (1(5 - 0)/1) = 5$		
			Bounds	
			var	
			lower	
			value	
			upper	
			x	
			y	
			s_1	

Figure 7.21: Deciding which variable to leave the basis and determining what will be its new value.

Sub-step 3.3 is to update the values of the basic variables and do a pivot operation. First, we go through all the constraints that contain the entering variable and update the value of the basic variables in each of these constraints. Then, we replace the leaving variable by the entering variable in the basis. Finally, we do linear arithmetic combinations to express the equations of the current problem in terms of the new set of basic variables, exactly like it is done in the primal simplex.

Knowing the variation of the entering variable value, we can obtain the new values of the basic variables by simple math. Similar to the previous formula for calculating the value entering variable, for each constraint we have:

$$v'_{bv} = v_{bv} + (c_{ev}(v_{ev} - v'_{ev})/c_{bv}) \quad (7.1)$$

In our example, we already know, from previous calculation, that s_1 will have value zero. After adding y to the basis and removing s_1 from it, it remains to do a linear combination in the goal to replace y by its new equivalent expression. The resulting tableau is shown in Figure 7.22.

We know that the goal will be satisfied when the value of its constant term reaches 0. In the last pivot this value went from 35 to 20. We got closer to a solution and to satisfy the new constraint, but we are not there yet and therefore we need to repeat the step three.

Now we have variables x and s_1 in the goal. We do the entering variable test to determine which variable will enter the basis. The test is shown in

	x	y	s_1	
y	-1	1	1	5
Max	4	0	-3	20

 $R'_2 = R_2 - 3R_1$

Bounds			
var	lower	value	upper
x	$-\infty$	0	∞
y	$-\infty$	5	∞
s_1	0	0	∞

Figure 7.22: The resulting tableau after the first pivot.

Figure 7.23. To increase the left side of the goal expression, the variable x needs to increase its value, while s_1 needs to decrease. In the test we see that s_1 is directly limited by its bound and cannot have its value decreased. On the other side, x is unbounded and can have its value increased as much as the constraints of the problem let, so x is chosen to enter the basis.

	x	y	s_1	
y	-1	1	1	5
Max	4	0	-3	20

Entering variable test

↓

$4(\infty - 0) = \infty$

↓

$-3(0 - 0) = 0$

Bounds			
var	lower	value	upper
x	$-\infty$	0	∞
y	$-\infty$	5	∞
s_1	0	0	∞

Figure 7.23: Deciding, between x and s_1 , which variable to enter the basis.

Next step is to determine by how much x is limited by the only constraint we have by now and see what will be the new value of x . Figure 7.24 shows the test. The test indicates that the value of x is not limited by the constraint where y is basic variable. That means that we could change its value to as much as we want.

	x	y	s_1	
y	-1	1	1	5
Max	4	0	-3	20

 $\implies v'_x = 0 + (1(5 - \infty) / -1) = \infty$

Leaving variable test

Bounds			
var	lower	value	upper
x	$-\infty$	0	∞
y	$-\infty$	5	∞
s_1	0	0	∞

Figure 7.24: Determining how much the value of x can increase.

In practice, we just change the minimum to make the goal satisfied. We know that to make the left side of the goal equal to the right side, we need to increase it by 20. If the coefficient of x in the goal is 4, then is enough to increase the value of x by 5.

If changing the value of the entering variable is enough to reach the goal then we do not need to pivot and change the basic variables. However, we still need to update the values of the basic variables.

Once the values are updated, we add the goal to the set of constraints by creating a new slack variable. After that, we are done with the new constraint and ready to receive another one.

In the example, the new value of x is 5, and that makes the new value of y equal to 10. The slack variable created is s_2 . Setting the lower bound of s_2 to 0, makes its coefficient equals to -1. Since the equation is balanced, the value of s_2 is initially 0. The resulting tableau can be seen in Figure 7.25.

	x	y	s_1	s_2			Bounds		
y	-1	1	1	0	5	var	lower	value	upper
s_2	4	0	-3	-1	20	x	$-\infty$	5	∞
						y	$-\infty$	10	∞
						s_1	0	0	∞
						s_2	0	0	∞

Figure 7.25: The final satisfying state of the tableau after the constraint $x + 3y \geq 35$ is processed.

We can see in the two dimension graph of Figure 7.26, the result of processing the constraint $x + 3y \geq 35$. We went from the solution $(x = 0, y = 0)$ to the solution $(x = 5, y = 10)$ passing through the point $(x = 0, y = 5)$. The highlighted area is the new feasible region, now considering the first two constraints.

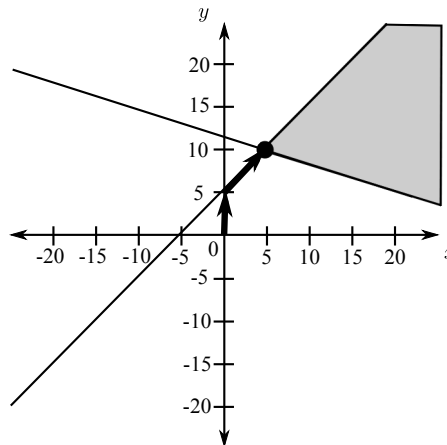


Figure 7.26: Resulting graph after processing and adding the constraint $x + 3y \geq 35$.

The new constraint to add is $x \geq 0$. There is only one variable which is already known, the variable x . x is not a basic variable, so the constraint is already normalized. Additionally the constraint is already satisfied since the value of x is 5.

It only remains to add this new constraint to the problem. As this is a special case, with only one variable, we choose to not add it to the set of constraint, but to change the lower bound of the variable x . Making this information available directly in the bounds of the variable is preferable as it makes the operations of the method faster and we also avoid to create extra slack variables.

The new state of the tableau does not change, only the bounds table. It can be seen in the Figure 7.27. The feasible region of the current set continues the same of the Figure 7.26 as well as the solution point.

	x	y	s_1	s_2	
y	-1	1	1	0	5
s_2	4	0	-3	-1	20

Bounds			
var	lower	value	upper
x	0	5	∞
y	$-\infty$	10	∞
s_1	0	0	∞
s_2	0	0	∞

Figure 7.27: The tableau and the value/bounds table after the constraint $x \geq 0$ is processed.

The next constraint to process is $y \leq 5$. y is a known variable and is in the basis. We normalize this constraint to replace y by its equivalent expression, so the constraint becomes $x - s_1 \leq 0$. The evaluation of the expression ($5 \leq 0$) indicates it is not satisfied, so we proceed the operations to change the values of the variables and satisfy the new constraint.

This time the left side of the inequality needs to decrease its value to reach the same amount of the right side. We could proceed by performing a minimization, doing just small modifications in the operations used in the maximization to have a coherent method. However, it is simpler to multiply the inequality by -1 and perform a maximization over the constraint $-x + s_1 \geq 0$. The current tableau, that we are going to work with, is shown in Figure 7.28.

	x	y	s_1	s_2	
y	-1	1	1	0	5
s_2	4	0	-3	-1	20
Max	-1	0	1	0	0

Bounds			
var	lower	value	upper
x	0	5	∞
y	$-\infty$	10	∞
s_1	0	0	∞
s_2	0	0	∞

Figure 7.28: The next goal was set to maximize $-x + s_1$ to reach 0.

Doing the entering and leaving variable test, it is decided that s_1 enters the basis and s_2 leaves the basis. The test can be seen in Figure 7.29. This time, the current solution was not improved, but the algorithm continues as

there is still room for improvement.

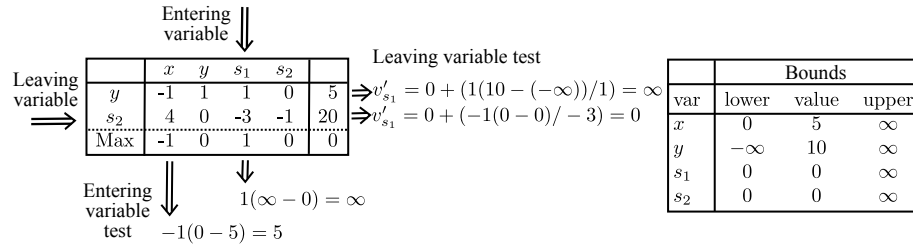


Figure 7.29: s_1 enters the basis in the place of s_2 .

After the variable s_1 enters the base, the next entering variable test determines that variable x should now enter, as it can be seen in Figure 7.30. The leaving variable test reveals that none of the constraints is limiting x , so it can be increased as much as it is necessary. We increase x to 20, the minimum necessary to satisfy the goal, and then update the value of the basic variables. The result can be seen in Figure 7.31.

Although the satisfying process of the constraint $y \leq 5$ was not direct as of the constraint $x \geq 0$, the constraint $y \leq 5$ is also a special case where there is only one variable. Therefore, we include this information not in the tableau, but in the bounds table. In fact, if we are processing a constraint and, at some moment, it contains only one variable, we can choose in the end to change the bound of this variable instead of including this constraint in the tableau.

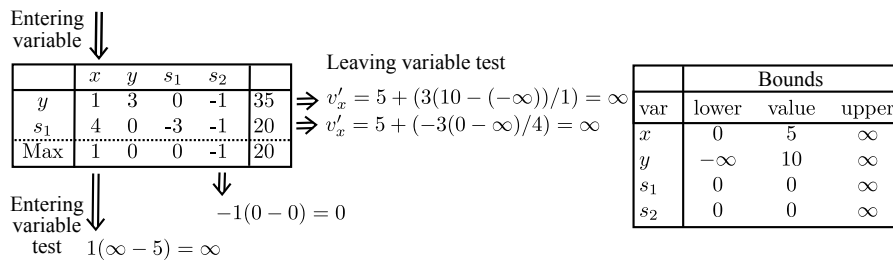


Figure 7.30: x would enter the basis, but since no other constraint is limiting its increase, x has will have it value changed to 20 and will remain out of the basis.

Geometrically, the result of processing the constraint $y \leq 5$ can be seen in Figure 7.32. The new cut in the feasible region, related to this constraint, made the solution point move from $(x = 5, y = 10)$ to $(x = 20, y = 5)$.

The last constraint to process in our problem is the equation $3x + 3y = 120$. There are no new variables. Normalizing the equation, we get $2x + s_2 =$

	x	y	s_1	s_2	
y	1	3	0	-1	35
s_1	4	0	-3	-1	20
Max	1	0	0	-1	20

Bounds			
var	lower	value	upper
x	0	20	∞
y	$-\infty$	5	∞
s_1	0	20	∞
s_2	0	0	∞

Figure 7.31: Tableau and values of the variables satisfying the goal.

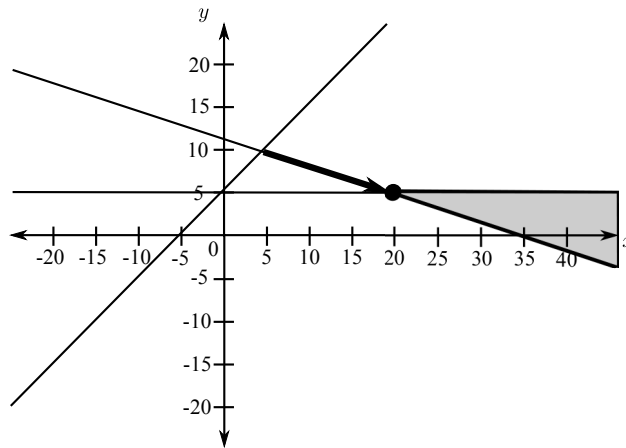


Figure 7.32: Resulting graph after processing $y \leq 5$.

85. The evaluation of the equation, $40 = 85$, indicates that the constraint is not satisfied. As in the inequality case, we are going to maximize the left side, initially $2x + s_2$, to reach the same value of the right side. Figure 7.33 shows the initial tableau.

	x	y	s_1	s_2	
y	1	3	0	-1	35
s_1	4	0	-3	-1	20
Max	2	0	0	1	85

Bounds			
var	lower	value	upper
x	0	20	∞
y	$-\infty$	5	5
s_1	0	20	∞
s_2	0	0	∞

Figure 7.33: The next goal was set to maximize $2x + s_2$ to reach 85.

Doing the entering and leaving variable test, we find out that x can have its value increased as much as necessary, as shown in Figure 7.34. We then set the value of x to 42.5 and propagate the change by updating the basic variables. The difference now is that as this constraint is not an inequality, we can choose one of its variable to be the basic variable instead of creating

a new one. The extra step is that we need to normalize the other constraints to remove the new basic variable. The final tableau is shown in Figure 7.35.

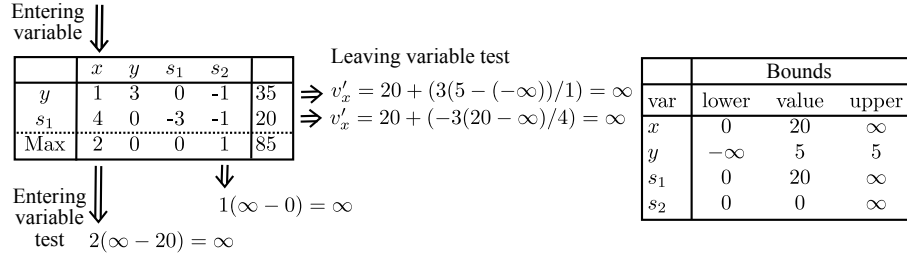


Figure 7.34: x is chosen to change the value.

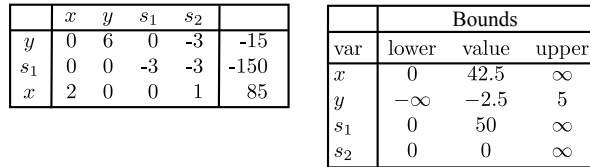


Figure 7.35: The final tableau, after the constraints $-x + y \leq 5$, $x + 3y \geq 35$, $x \geq 0$, $y \leq 5$ and $3x + 3y = 120$ were processed.

The new graph representing the problem and the solution is shown in Figure 7.36. Notice that after the last equation was added the feasible region got very restricted (in this graph, represented only by a line segment).

7.4 The unsatisfiable case

We have seen all the particular cases of the algorithm where the final status is satisfiable. The remaining situation is when a new constraint makes the problem unsatisfiable. Consider that a new constraint arrives to the same problem:

$$x \geq 50$$

We proceed as before. Normalizing the constraint we get $-s_2 \geq 15$ which we set as the goal. s_2 is currently the only candidate variable to change the value to satisfy the goal. It needs to have its value decreased. However, doing the entering test variable, we get that s_2 cannot decrease its value anymore. Its value is directly limited by its lower bound, as it shows the test in Figure 7.37.

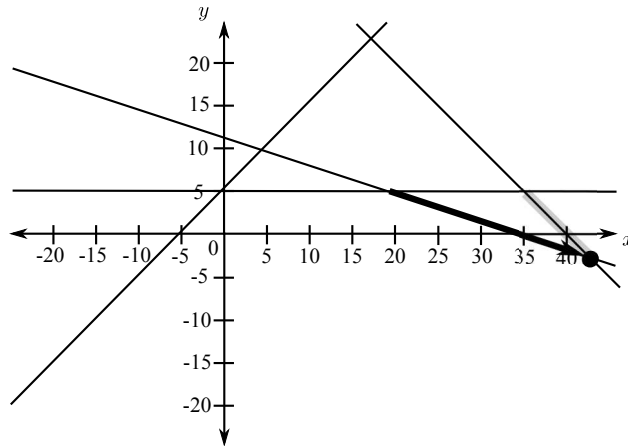


Figure 7.36: The final graph in two dimensions, after the constraints $-x + y \leq 5$, $x + 3y \geq 35$, $x \geq 0$, $y \leq 5$ and $3x + 3y = 120$ were processed.

	x	y	s_1	s_2	
y	0	6	0	-3	-15
s_1	0	0	-3	-3	-150
x	2	0	0	1	85
Max	0	0	0	-1	15

\Downarrow
 Entering variable test $-1(0 - 0) = 0$

Bounds			
var	lower	value	upper
x	0	42.5	∞
y	$-\infty$	-2.5	5
s_1	0	50	∞
s_2	0	0	∞

Figure 7.37: The only variable in the goal, s_2 , cannot have its value changed to satisfy the goal.

If all of the variables in the goal cannot have their value changed because they are directly limited by their bounds, it means we cannot satisfy the new constraint and therefore the problem is unsatisfiable.

7.5 Generating the conflict set

The reason why the problem is unsatisfiable is because the variables in the goal cannot have their value changed to make the goal satisfiable. That is because all these variables are directly limited by their bounds. We want to return the original set of constraints that reflects this explanation.

First, we save apart all linear arithmetic combinations we do with the constraints during the algorithm, like in the normalization and the pivot operations. We keep associated to each of the constraints a linear expression that shows what were the linear combinations done so far to obtain this constraint.

An illustrative example is shown in Figure 7.38. It is an extended tableau of Figure 7.37 that also shows the explanation of the linear combinations. For instance, the constraint in the first row of the tableau ($6y - 3s_2 = -15$, or equivalently, $6y \geq -15$) is obtained when we multiply the second user constraint ($x + 3y \geq 35$) by 3 and subtract the result by the fifth user given constraint ($3x + 3y = 120$).

When we normalize the sixth constraint $x \geq 50$, by multiplying this new constraint by 2 and subtracting by the third row of the tableau, the same is done with the side explanation. The explanation of the sixth constraint becomes $2C_6 - (-C_2 + C_5)$.

	x	y	s_1	s_2		Explanation
y	0	6	0	-3	-15	$3C_2 - C_5$
s_1	0	0	-3	-3	-150	$-3C_1 + 3C_2 - 2C_5$
x	2	0	0	1	85	$-C_2 + C_5$
Max	0	0	0	-1	15	$C_2 - C_5 + 2C_6$

	Bounds			Explanation	
var	lower	value	upper	lower	upper
x	0	42.5	∞	C_3	\emptyset
y	$-\infty$	-2.5	5	\emptyset	C_4
s_1	0	50	∞	\emptyset	\emptyset
s_2	0	0	∞	\emptyset	\emptyset

Figure 7.38: Extended version of the tableau with the linear explanation of the combinations.

The second side task we do to build the conflict set is to save also the explanation of why the bound has some value different from $+\infty$ or $-\infty$. For instance, in the Figure 7.38, x has lower bound equal to zero because of the third constraint given by the user, $x \geq 0$.

We do not need to worry about lower bound explanations of slack variables because they are crafted internal information. The reason why they have lower bound equals to zero comes from the transformation of the inequalities into equations, and not because of the user given constraints.

So, for constructing the conflict set, if we know that the problem is unsatisfiable because we cannot modify the values of the variables in the goal to do a consistent evaluation, we simply construct the conflict set by collecting the constraints in the explanation that led to goal expression plus the constraints that explain the limitation of the bounds of the variables in the goal. As we are maintaining this information updated, constructing the conflict set is a straightforward task.

In the case of our last example, the conflict set is $\{C_2, C_5, C_6\}$ which is $\{x + 3y \geq 35, 3x + 3y = 120, x \geq 50\}$. That is all the information that the decision procedure needs to return. However, notice (although not required by the decision procedure) that summing the exact explanation of the goal

and the bounds we can precisely find the inconsistency:

$$\begin{array}{r}
 (x + 3y \geq 35) \\
 + \quad -1 \times \quad (3x + 3y = 120) \\
 + \quad 2 \times \quad (x \geq 50) \\
 \hline
 (x + 3y \geq 35) \\
 + \quad (-3x - 3y = -120) \\
 + \quad (2x \geq 100) \\
 \hline
 0 \geq 15
 \end{array}$$

7.6 Backtracking

One of the good aspects of this original algorithm is that we do not need to be in the vertex of a simplex before running it. This property simplifies considerably the backtrack.

The backtrack needs to return the decision procedure to a state equivalent to one before the constraint was added to the problem. In the case of our proposed algorithm, two states are equivalent if they are both satisfiable or both unsatisfiable, and all the information kept in the tables comes from linear combinations of the same set of constraints. It is easy to fulfill this requirement to backtrack as it is not necessary to return to the exactly same solution as before, neither have to save the values of the variables to recover them later.

When going from a satisfiable to an unsatisfiable solution, simply removing the last constraint will return the problem back to a satisfiable solution. In the case of going from a satisfiable solution to another satisfiable solution, removing the last constraint will not violate the current solution. That happens because when the algorithm is looking for a solution, it always respects the previously added constraints. Therefore, if we simply remove the last constraint, we are sure to return to a satisfiable solution.

We can see a graphical example of an arbitrary problem in Figure 7.39. The state goes from satisfiable to unsatisfiable and then backtrack is performed.

There is just one major detail while backtracking. Due to the combinations performed during the algorithm, there may remain some trace of the constraint that we want to remove in the other constraints. See, for instance, that if we remove the sixth and the fifth constraints in Figure 7.38, it still remains a trace of the fifth constraint C_5 in the first and second rows.

The solution for this is simple though. Before removing a constraint C_k , we go through all the other constraints and remove any trace they may contain of C_k by doing a linear combination as shown by the *explanation*. So in the example of Figure 7.38, before removing C_5 , we first subtract C_5 from the first row, and two times C_5 from the second row.

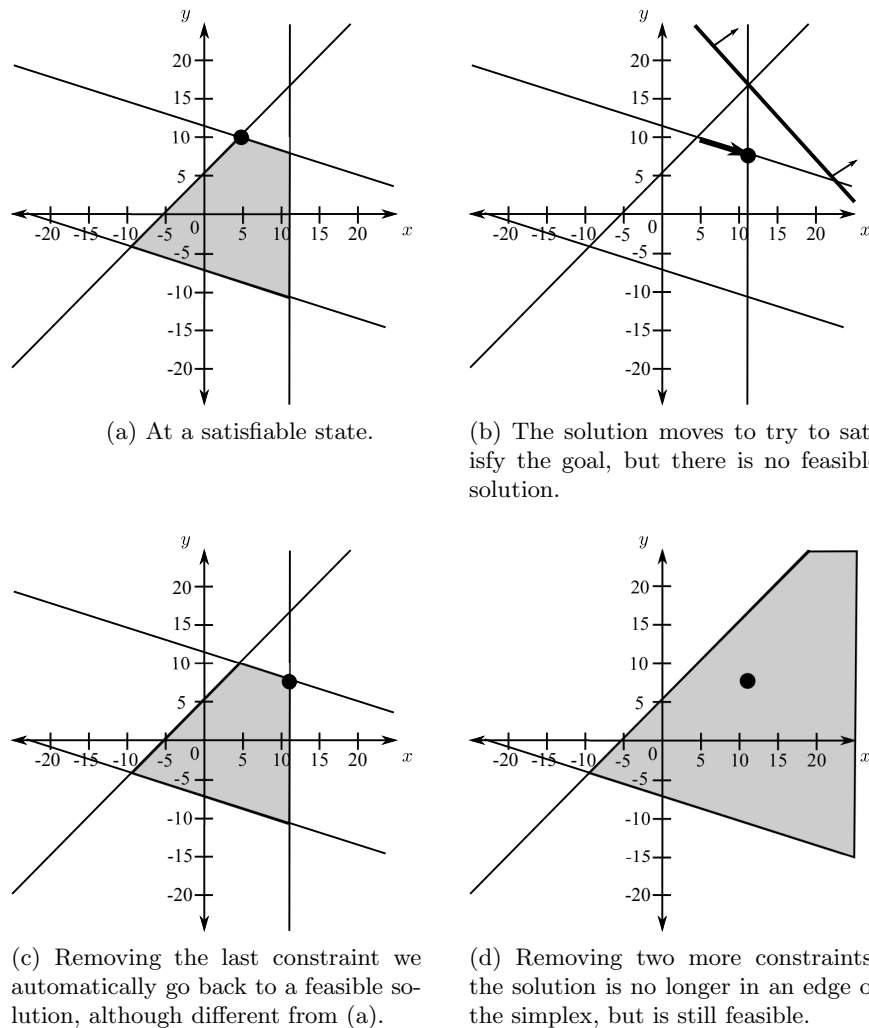


Figure 7.39: Backtrack in action.

The remaining tasks while backtracking are direct, like restoring the previous upper and lower bound, or undoing the registration of a variable (if it happened to be created by the removed constraint).

7.7 Equality generation

Complete strategies for generating all equalities between variables are costly. But some non expensive search can be done to find some equalities in a few cases.

A low cost deduction can be done when two variables have the same value and they cannot have another value because their lower bound are equal to

their upper bound. This is the easiest way of detecting some equalities.

A not so low cost deduction can be done when comparing equations. If isolating two variables on the left side of two different equations, one gets the same right side in both equations, then the two variables are equal to each other. For instance, we can deduce that $x = y$ from the two following constraints:

$$\begin{aligned}x &= 3z - 4w + 5 \\y &= 3z - 4w + 5\end{aligned}$$

A high cost, but complete way of detecting equalities is to ask information to the simplex. In a satisfiable state of the problem, for each pair of variables (x, y) that have the same value, we set as the goal (in two different runs) first $x > y$ and then $x < y$. If in both cases the goal cannot be satisfied, we can conclude that $x = y$.

These are some examples of equality detection that can be applied. The first two strategies are applied in some SMT-solvers, like described by de Moura and Nikolaj [25]. However, so far in our solver `veriT`, we only apply model-equality generation (as explained in the next section) to obtain completeness.

7.8 Model-equality generation

Model-equalities are used to obtain completeness for a combination of theories like explained in 4. Generating model-equalities is very simple and direct in our case. We just look at the values of the variables and create model-equalities between the variables that have the same value.

Because of the nature of the algorithm, the values in our model are naturally more spread than, for instance, the values in the model of the difference logic decision procedure. This makes the decision procedure for linear arithmetic generate fewer model-equalities.

That is how it is implemented in our solver `veriT`. However, as the algorithm works even if the solution is not in the vertex of the simplex, in the case it is wished to reduce the number of model-equalities, some heuristics could be used to change the values of the variables while respecting the constraints, but this remains a task to experiment.

7.9 Disequalities and strict inequalities

Disequalities cannot be incorporated into the tableau, so they are stored and checked later. The decision procedure for linear arithmetic can handle the disequalities in the same way as the difference logic decision procedure presented in Section 6.2.5.

The decision procedure checks for the model-equalities and generates lemmas to modify the current model whenever there are conflicts between the model and disequalities. For instance, if in our model we have $x = y$ and there exists the disequality $x \neq y$, then the decision procedure would generate the lemma $x \neq y \implies x < y \vee x > y$ to incorporate in the SAT-solver, that will in the next iteration modify the model.

Handling strict inequalities is done exactly like in the difference logic case, explained in details in Section 6.1.1. We replace constraints like $x + y < c$ to $x + y \leq (c - \delta)$ and change the way we represent the numbers to be able to do mathematic operations with delta.

7.10 Integer variables

The way it was presented so far, the decision procedure is complete for reals and can be used in combination with other decision procedures in an SMT-solver. However it is not the case when there are integer variables.

The general strategy of simplex algorithms to handle integer variables is to use branch-and-cut techniques [45], which is a mix of branch-and-bound with cutting planes [36]. We run the simplex and whenever the final solution gives a real value to an integer variable, cutting planes and branch-and-bound techniques are alternately applied to eliminate this solution and get a new one that hopefully will have an integer value.

The basic idea of cutting planes techniques is to generate a constraint that will eliminate the current real solution and possible a few other ones, but will not eliminate any valid integer solution. If the new solution is not integer, it will continue to generate constraints (cuts) until an integer solution is found.

Branch and bound works by creating case splits at integer variables with real values, invalidating the current solution. As a simple example, let x be an integer variable with invalid value 1.5, branch and bound will split and try two scenarios, one adding the constraint $x \leq 1$ and other adding the constraint $x \geq 2$. If in one of the scenarios an integer solution is found, the solution is also valid to the original problem.

While branch and bound is simple, it may have termination issues when variables are unbounded, so it is not enough to be used alone. Cutting planes, on the other side, is a complete strategy, but it is not trivial to generate the cuts. Modern strategies try to play both techniques at the same time to find an integer solution.

A complete strategy for integer variables remains as a work to do in our solver veriT. We still need to think about a way to adapt cutting planes to our simplex variation. So far we have implemented the branch and bound strategy and the *greatest common divisor verification*. Together, they very often help to find an integer solution or identify that there are none.

The greatest common divisor (gcd) verification can be applied on single constraints where all variables are integers. Assuming that all coefficients in the constraint are integer, we know that this constraint cannot have an integer solution if the constant term is not divisible by the greatest common divisor of the coefficients of the variables⁵. Doing this test, for instance, it is easy to know that an expression like $2x + 4y = 5$ has no integer solution because 5 cannot be divided by $gcd(2, 4)$, which is 2.

7.11 Conclusion and future work

We presented in this chapter an original variation of the simplex algorithm which is incremental, very flexible and can be used to build a decision procedure for linear arithmetic. We presented also the extra requirements necessary to integrate this decision procedure in an SMT-solver. The decision procedure can be used in combination with others in an SMT-solver and is complete in the case of real variables.

There is still some remaining work to do. To get completeness when there are integer variables, we need to explore the remaining idea of generating cutting planes. It also remains to experiment mixing generation of equalities and model-equalities, and see how much we can gain in efficiency. Finally, theory propagation also remains a work to do.

On the veriT side, there are some remaining tasks to do too. We need smarter and more aggressive pre and inter processing strategies to simplify the problem and reduce the number of variables given to linear arithmetic decision procedure.

Also on the computational side, we can rethink the problem to remove the necessity to update and maintain the full table at each step. The algorithm was explained in the form of tableau. Although it is the easiest way to explain and it works well for small and middle size problems, the tableau form is not computationally the best for large problems. As the problem increases in the number of variables, originally sparse problems tend to become dense. When we are maintaining a big dense tableau we are also doing many operations that are not actually necessary. If we pay attention to the operations we do to determine if the algorithm can continue, what variable to enter or leave, etc. we are only looking at a few rows and columns. In the revised simplex method [70], instead of maintaining the tableau updated, it calculates these few rows and columns every time it is necessary. The calculation is heavy and it will not be explained here, but it is possible to calculate these rows and columns only knowing the original tableau and the variables that are currently in the basis. Commercial simplex implementations are based on the revised simplex.

⁵In number theory, this is a generalization of Bézout's identity, which is a linear diophantine equation, and that was proved by Étienne Bézout (1730-1783).

Comparison with other decision procedures is delicate because the implementations of linear arithmetic decision procedures available are usually integrated in SMT-solvers and therefore they have many features that interfere in the results. Features like theory propagation and also stronger arithmetic pre-processing are missing in *veriT* which gives a disadvantage and makes it hard to do a fair comparison between decision procedures. Anyway, with our current implementation in a few crafted experiments, it was possible to observe that it has good performance when the problem does not increase the density during the execution of the algorithm. It scales very well for the number of the constraints, being able to receive a few thousands, but not as good when the number of variables increase, being limited by about a hundred. We believe that implementing the remarks from the previous paragraphs can make *veriT* a top performance solver for linear arithmetic problems.

Chapter 8

Conclusion

In this thesis, we presented methods and techniques to build cooperative decision procedures in an original combination framework used in SMT-solvers. The methods were for the arithmetic fragments of difference logic and linear arithmetic but most of the concepts can be applied to other theories.

In Chapter 3, we introduced an essential component of SMT-solvers, the SAT-solver. We showed the basic algorithm which most of the modern SAT-solvers are based on, a short description of advanced techniques, and an example of how to encode a problem in SAT. The second part of this chapter presented the SMT-solvers. We described its basic architecture and how it works, ending with an example of how to encode in SMT.

In Chapter 4, we demonstrated how decision procedures from different theories can be combined inside the SMT-solvers. We explained three ways of combining decision procedures based on the Nelson and Oppen combination framework. We reveal the difficulties of combining non-convex theories, and then we proposed an alternative method that extends the Nelson and Oppen combination framework, using *model-equalities* to make the cooperation between decision procedures easier and more precise.

In Chapter 5, we detailed the requirements to build efficient and cooperative decision procedures for SMT-solvers. Most of the motivations of these requirements come from the SAT-solver and the combination framework based on Nelson and Oppen explained in Chapters 3 and 4.

In Chapter 6, we introduced difference logic and a way of representing this theory using graphs. We demonstrated some interesting properties for the graph representation that help us to understand the algorithms to build a decision procedure for difference logic. Then, we explained all the details necessary to build a complete decision procedure for real and integer difference logic, fulfilling all the requirements seen before. It was also demonstrated details of how to generate model-equalities for difference logic and obtain a decision procedure that can also be used in a combination

framework inside an SMT-solver.

In Chapter 7, we presented an original variation of the simplex algorithm which is incremental, very flexible and can be used to build a decision procedure for linear arithmetic. We provided also the extra functionalities for the requirements necessary to integrate this decision procedure in an SMT-solver. The decision procedure can be used in combination with others in an SMT-solver to produce complete results in the case of real variables.

Practically all the work presented in this thesis was implemented in our SMT-solver `veriT` which is open-source, under the BSD license, and available for download at www.verit-solver.org. The arithmetic module, which includes the decision procedures for difference logic and linear arithmetic, is implemented using the C language and has about 9,000 lines of code. The source is available with the `veriT` solver.

A summary of the contributions of my thesis includes:

- An extension of the Nelson and Oppen combination framework to combine decision procedures using model-equalities.
- A description of how to implement model-equalities for difference logic theory.
- An open-source implementation of a difference logic decision procedure, used by the `veriT` SMT-solver.
- An incremental algorithm based on the simplex used to check the satisfiability of linear arithmetic constraints.
- The fulfillment of the requirements to build a complete decision procedure for real linear arithmetic, and partial for integers.
- An open-source implementation of a linear arithmetic decision procedure, used by the `veriT` SMT-solver.

There are also remaining things to do and many research directions that came from the results of this thesis. Varying from a few weeks work to several months tasks, here are some of them:

- It would be interesting to see in practice how well our proposed combination framework works with a variety of theories. We would like to combine more and different theories and see how easy it would be to generate *model-equalities* and how good would be the sharing of information.
- *Non-linear arithmetic* is present in some verification problems. Despite of complexity issues of the real case and undecidability of the integer

case, it would be desirable to have also a (semi¹-)decision procedure for non-linear arithmetic. However, the difficulties of this theory makes it a large task.

- Another research direction is to experiment cooperating different arithmetic decision procedures to increase the general efficiency of solving problems. Identifying ways of using faster decision procedures to help slower ones can be an interesting research topic.
- Our implementation of integer linear arithmetic is still not complete. Adapting our algorithm to incorporate *cutting planes* techniques is probably the solution.
- A huge part of the effort of implementing the simplex efficiently comes from designing smart data structures. Applying the techniques from the *revised simplex method* also boosts the simplex efficiency. This is an important next step in the development of our variation of the simplex algorithm.
- There are only a few ideas about *theory propagation*. Theory propagation can help a lot solving some problems and it is definitely one of the priorities for the near future.

¹A semi-decision procedure is able to deduce satisfiability or unsatisfiability but is not complete for both cases.

Bibliography

- [1] Smtcomp webpage. www.smtcomp.org, 2010.
- [2] Greg J. Badros and Alan Borning. The cassowary linear arithmetic constraint solving algorithm: Interface and implementation. Technical report, ACM transactions on computer human interaction, 1998.
- [3] Clark Barrett, Morgan Deters, Albert Oliveras, and Aaron Stump. Design and results of the 3rd annual satisfiability modulo theories competition (SMT-COMP 2007). *International Journal on Artificial Intelligence Tools*, 17(4):569–606, 2008.
- [4] Clark Barrett, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Splitting on demand in SAT modulo theories. In Miki Hermann and Andrei Voronkov, editors, *Proc. 13th Int'l Conf. on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, volume 4246 of *Lecture Notes in Computer Science*, pages 512–526. Springer, 2006.
- [5] Stump A. Tinelli C. Barrett, C. The SMT-LIB standard: Version 2.0. Technical report, Department of Computer Science, The University of Iowa, 2010.
- [6] Richard H. Bartels and Gene H. Golub. The simplex method of linear programming using LU decomposition. *Commun. ACM*, 12:266–268, May 1969.
- [7] Roberto J. Bayardo. Using CSP look-back techniques to solve real-world SAT instances. pages 203–208. AAAI Press, 1997.
- [8] Richard Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958.
- [9] D. Le Berre and L. Simon (Organizers). SAT 2009 competition. <http://www.satcompetition.org/2009/>, 2009.
- [10] Daniel Le Berre and Laurent Simon. Fifty-five solvers in vancouver: The SAT 2004 competition. In Holger H. Hoos and David G. Mitchell, editors, *SAT (Selected Papers)*, volume 3542 of *Lecture Notes in Computer Science*, pages 321–344. Springer, 2004.

- [11] Frédéric Besson. On using an inexact floating-point LP solver for deciding linear arithmetic in an SMT solver. *8th International Workshop on Satisfiability Modulo Theories*, 2010.
- [12] Per Bjesse, Tim Leonard, and Abdel Mokkedem. Finding bugs in an alpha microprocessor using satisfiability solvers. pages 454–464. Springer-Verlag, 2001.
- [13] R. Bland. New finite pivoting rules for the simplex method. *Mathematics of Operations Research*, 2:103–107, 1977.
- [14] Thomas Bouton, Diego Caminha B. de Oliveira, David Déharbe, and Pascal Fontaine. veriT: An open, trustable and efficient SMT-solver. In Renate A. Schmidt, editor, *CADE*, volume 5663 of *Lecture Notes in Computer Science*, pages 151–156. Springer, 2009.
- [15] Thierry Boy de la Tour. An optimality result for clause form translation. *J. Symb. Comput.*, 14:283–301, October 1992.
- [16] Marco Bozzano, Roberto Bruttomesso, Ro Cimatti, Tommi Junttila, Silvio Ranise, Peter Van Rossum, and Roberto Sebastiani. Efficient satisfiability modulo theories via delayed theory combination. In *In Proc. CAV 2005, volume 3576 of LNCS*, pages 335–349. Springer, 2005.
- [17] Roberto Bruttomesso, Ro Cimatti, Anders Franzen, Alberto Griggio, and Roberto Sebastiani. Delayed theory combination vs. nelson-oppen for satisfiability modulo theories: A comparative analysis. In *In Proc. LPAR 06, volume 4246 of LNAI*, pages 527–541. Springer, 2006.
- [18] Stephen A. Cook. The complexity of theorem-proving procedures. In *STOC '71: Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158, New York, NY, USA, 1971. ACM.
- [19] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition*. The MIT Press and McGraw-Hill Book Company, 2001.
- [20] G. B. Dantzig, A. Orden, and P. Wolfe. The generalized simplex method for minimizing a linear form under linear inequality restraints. *Pacific J. Math.*, 5:183–195, 1955.
- [21] George Dantzig. *Linear Programming and Extensions*. Princeton University Press, August 1998.
- [22] George B. Dantzig and B. Curtis Eaves. Fourier-Motzkin elimination and its dual. *Journal of Combinatorial Theory (A)*, 14:288–297, 1973.
- [23] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5:394–397, July 1962.

- [24] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7:201–215, July 1960.
- [25] Leonardo de Moura and Nikolaj Bjørner. Model-based theory combination. *Electronic Notes in Theoretical Computer Science*, 198(2):37–49, 2008.
- [26] Diego Caminha B. de Oliveira, David Déharbe, and Pascal Fontaine. Combining decision procedures by (model-)equality propagation. *Electron. Notes Theor. Comput. Sci.*, 240:113–128, 2009.
- [27] Diego Caminha Barbosa de Oliveira. Deciding difference logic in a Nelson-Oppen combination framework. Master’s thesis, Federal University of Rio Grande do Norte, Natal, Brazil, 2007.
- [28] David Detlefs, Greg Nelson, James, and B. Saxe. Simplify: A theorem prover for program checking. Technical report, J. ACM, 2003.
- [29] Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.
- [30] Bruno Dutertre and Leonardo de Moura. A fast linear-arithmetic solver for DPLL(T). In Thomas Ball and Robert Jones, editors, *Computer Aided Verification*, volume 4144 of *Lecture Notes in Computer Science*, pages 81–94. Springer Berlin / Heidelberg, 2006.
- [31] Bruno Dutertre and Leonardo De Moura. Integrating simplex with DPLL(T). Technical report, CSL, SRI INTERNATIONAL, 2006.
- [32] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing*, volume 2919 of *Lecture Notes in Computer Science*, pages 333–336. Springer, 2003. 6th International Conference, SAT 2003.
- [33] Germain Faure, Robert Nieuwenhuis, Albert Oliveras, and Enric Rodríguez-Carbonell. SAT modulo the theory of linear arithmetic: Exact, inexact and commercial solvers. In Hans Kleine Büning and Xishun Zhao, editors, *Theory and Applications of Satisfiability Testing, SAT 2008*, volume 4996 of *Lecture Notes in Computer Science*, pages 77–90. Springer Berlin / Heidelberg, 2008.
- [34] L. R. Ford, Jr. and D. R. Fulkerson. *Flows in Networks*. Princeton University Press, Princeton, 1962.
- [35] Carla P. Gomes, Bart Selman, Ken McAloon, and Carol Tretkoff. Randomization in backtrack search: Exploiting heavy-tailed profiles for solving hard scheduling problems. In *AIPS*, pages 208–213, 1998.

- [36] R. E. Gomory. An algorithm for integer solutions to linear programs. In R. L. Graves and P. Wolfe, editors, *Recent Advances in Mathematical Programming*, pages 269–302. McGraw-Hill, New York, 1963.
- [37] Robert G. Jeroslow and Jinchang Wang. Solving propositional satisfiability problems. *Annals of Mathematics and Artificial Intelligence*, 1:167–187, 1990.
- [38] Henry A. Kautz and Bart Selman. Pushing the envelope: Planning, propositional logic and stochastic search. In *AAAI/IAAI, Vol. 2*, pages 1194–1201, 1996.
- [39] Haluk Konuk and Tracy Larrabee. Explorations of sequential atpg using Boolean satisfiability. In *In 11th VLSI Test Symposium*, pages 85–90, 1993.
- [40] Daniel Kroening and Ofer Strichman. *Decision Procedures: An Algorithmic Point of View*. Springer Publishing Company, Incorporated, 1 edition, 2008.
- [41] Oliver Kullmann. Fundamentals of branching heuristics. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 205–244. IOS Press, 2009.
- [42] Shuvendu K. Lahiri and Madanlal Musuvathi. An efficient nelson-oppen decision procedure for difference constraints over rationals. *Electr. Notes Theor. Comput. Sci.*, 144(2):27–41, 2006.
- [43] J. P. Marques-Silva and K. A. Sakallah. GRASP: A New Search Algorithm for Satisfiability. In *International Conference on Computer-Aided Design (ICCAD'96)*, pages 220–227. IEEE Press, 1996.
- [44] João P. Marques-silva and Karem A. Sakallah. Grasp: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48:506–521, 1999.
- [45] J. E. Mitchell. Branch-and-cut algorithms for combinatorial optimization problems. In P. M. Pardalos and M. G. C. Resende, editors, *Handbook of Applied Optimization*, pages 65–77. Oxford University Press, January 2002.
- [46] David Monniaux. On using floating-point computations to help an exact linear arithmetic decision procedure. In *Computer-aided verification (CAV)*, volume 5643 of *Lecture Notes in Computer Science*, pages 570–583. Springer Verlag, 2009.

- [47] Steven S. Morgan. A comparison of simplex method algorithms. Master's thesis, University of Florida, 1997.
- [48] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *ANNUAL ACM IEEE DESIGN AUTOMATION CONFERENCE*, pages 530–535. ACM, 2001.
- [49] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, pages 530–535, 2001.
- [50] Alexander Nadel and Vadim Ryvchin. Assignment stack shrinking. In *SAT*, pages 375–381, 2010.
- [51] George Ciprian Necula. *Compiling with proofs*. PhD thesis, Pittsburgh, PA, USA, 1998. AAI9918593.
- [52] G. Nelson and D. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, 1979.
- [53] G. Nelson and D. Oppen. Fast decision procedures based on congruence closure. *Journal of the ACM*, 27(2):356–364, 1980.
- [54] P. Niebert, M. Mahfoudh, E. Asarin, M. Bozga, N. Jain, and O. Maler. Verification of timed automata via satisfiability checking. In *Proc. Formal Techniques in Real-Time and FaultTolerant Systems FTRTFT'02, volume 2469 of Lecture Notes in Computer Science*, pages 225–244, 2002.
- [55] Robert Nieuwenhuis and Albert Oliveras. DPLL(T) with exhaustive theory propagation and its application to difference logic. In *In CAV'05 LNCS 3576*, pages 321–334. Springer, 2005.
- [56] D. Oppen. Complexity, convexity and combinations of theories. *Theoretical Computer Science*, 12(3):291–302, November 1980.
- [57] C. Sinz (Organizer). SAT-race 2010. <http://baldur.iti.uka.de/sat-race-2010/>, 2010.
- [58] S. Ranise and C. Tinelli. The SMT-LIB standard: Version 1.2. Technical report, Department of Computer Science, The University of Iowa, 2006.
- [59] Silvio Ranise and Cesare Tinelli. The Satisfiability Modulo Theories Library SMT-LIB). www.SMT-LIB.org, 2006.

- [60] J. K. Reid. A sparsity-exploiting variant of the bartels-golub decomposition for linear programming bases. *Mathematical Programming*, 24:55–69, 1982. 10.1007/BF01585094.
- [61] H. Rueß and N. Shankar. Solving linear arithmetic constraints. Technical Report Technical Report SRI-CSL-04-01, SRI International, 2004.
- [62] S. J. Russell and P. Norvig. *Artificial Intelligence: A modern Approach*. publisher, 2nd edition, 2002.
- [63] João P. Marques Silva. The impact of branching heuristics in propositional satisfiability algorithms. In *Proceedings of the 9th Portuguese Conference on Artificial Intelligence: Progress in Artificial Intelligence*, EPIA '99, pages 62–74, London, UK, 1999. Springer-Verlag.
- [64] Niklas Sörensson and Armin Biere. Minimizing learned clauses. In *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing*, SAT '09, pages 237–243, Berlin, Heidelberg, 2009. Springer-Verlag.
- [65] Richard M. Stallman and Gerald J. Sussman. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence*, 9(2):135 – 196, 1977.
- [66] P.R. Stephan, Robert K. Brayton, and Alberto L. Sangiovanni-Vincentelli. Combinational test generation using satisfiability. Technical Report UCB/ERL M92/112, EECS Department, University of California, Berkeley, 1992.
- [67] R. E. Tarjan. Depth first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, June 1972.
- [68] Cesare Tinelli and Mehdi Harandi. A new correctness proof of the Nelson-Oppen combination procedure. In *Frontiers of Combining Systems, volume 3 of Applied Logic Series*, pages 103–120. Kluwer Academic Publishers, 1996.
- [69] Miroslav N. Velev. Effective use of boolean satisfiability procedures in the formal verification of superscalar and vliw microprocessors. In *Journal of Symbolic Computation*, pages 226–231, 2001.
- [70] Harvey M. Wagner. A Comparison of the Original and Revised Simplex Methods. *OPERATIONS RESEARCH*, 5(3):361–369, 1957.
- [71] Stephen Warshall. A theorem on Boolean matrices. *J. ACM*, 9:11–12, January 1962.

- [72] Hantao Zhang and Jieh Hsiang. Solving open quasigroup problems by propositional reasoning. In *In Proceedings of the International Computer Symp*, 1994.
- [73] L. Zhang, C.F. Madigan, M.W. Moskewicz, and S. Malik. Efficient conflict driven learning in Boolean satisfiability solver. In *Proc. Int'l Conf. on Computer Aided Design (ICCAD)*, pages 279–285, 2001.
- [74] Lintao Zhang, Conor F. Madigan, Matthew H. Moskewicz, and Sharad Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design, ICCAD '01*, pages 279–285, Piscataway, NJ, USA, 2001. IEEE Press.