Représentation d'ensembles de valeurs numériques en vérification de programme

Rapport de Stage

METIVIER Hugo
Stage de Master 2 Recherche Informatique
Sous la direction de Bertrand Jeannet

Table des matières

1 Principe d'analyse d'accessibilité **5**
1.1 Modèle de programme et Système de transition 5
1.2 Propriété d'accessibilité 6
1.3 Résolution de l'équation de point fixe 7

2 Interprétation abstraite **9**
2.1 Approximation statique 9
2.2 Approximation dynamique 10
2.3 Exemple : L'analyse d'intervalle 10
2.3.1 Définition 10
2.3.2 Exemple d'analyse d'intervalle sur un programme 12
2.4 Construction de domaines abstraits 13
2.4.1 Réduction et forme canonique 13
2.4.2 Produit 13

3 Exemples de domaines abstraits pour les variables numériques **14**
3.1 Zones et Octogones 14
3.2 2 variables par équation 15
3.3 Polyèdres convexes 16
3.4 Les templates 16
3.5 Discussion 17

4 Problématique du stage **19**
4.1 Principe 19
4.2 Exemples 19
4.3 Produit réduit d'octogones 20
4.4 Choix des paquets 20

5 Le treillis des Octogones en détail **21**
5.1 Les zones 21
5.2 Représentation d'un octogone 22
5.3 Clôture et forme canonique 23

6 Le treillis des Octogones creux **24**
6.1 Représentation 24
6.2 Opération de réduction 25
6.2.1 Test de vacuité 25
6.2.2 Minimisation des contraintes : propriétés utilisées 26
6.2.3 Algorithme de réduction 31

6.3 Comparatif Octogone/Octogone creux 34

7 Opérations abstraites des Octogones creux 35

7.1 Union 35
7.2 Intersection 35
7.3 Inclusion 36
7.4 Test d'égalité 36
7.5 Projection 36
7.6 Evaluation de la borne supérieure d'une expression 36
7.7 Affectation 37
7.8 Intersection avec une condition 37

Introduction

La vérification de programmes est un enjeu dont l'importance n'est plus à souligner. Les recherches dans ce domaine se sont développées dans diverses directions, mais principalement basées, d'une part, sur la preuve formelle - souvent interactive -, d'autre part sur l'exploration exhaustive de modèles finis des programmes (model-checking), et enfin l'analyse statique, méthode approchée mais entièrement automatique.

Les logiciels les plus critiques, du point de vue de la sûreté, sont les logiciels embarqués temps réel. Dans ce domaine, comme dans beaucoup d'autres, la complexité croissante des applications rend irréaliste l'application des méthodes de preuve interactive. Par ailleurs, même si l'on se restreint aux noyaux de contrôle de tels systèmes, il est rare que l'on puisse en extraire automatiquement un modèle fini suffisamment précis : la raison en est que même le contrôle fait appel à des variables numériques (par exemple, des compteurs de temps, d'événements, ...) dont le comportement est significatif dans le respect des propriétés cruciales du système. C'est pourquoi nous nous intéressons ici à la vérification automatique de programmes manipulant des variables numériques, et donc de programmes à nombre infini d'états. Dans ce stage, l'objectif est de valider des propriétés sur les variables numériques de ce type de programmes qui peuvent se ramener à des problèmes d'accessibilité d'états.

Cet objectif se heurte à l'indécidabilité de la vérification de programmes généraux. On est ainsi amené à se contenter de vérification approchée : le résultat fourni sera soit "la propriété est satisfaite", soit "je ne sais pas". Dans ce domaine, les principales approches relèvent de l'interprétation abstraite, une théorie générale proposée dès 1977 [CC77]. Les invariants linéaires construits par l'analyse sont utilisés, en vérification, pour montrer l'inaccessibilité de certains points de contrôle. En compilation, ils peuvent servir à déterminer statiquement l'absence d'erreur à l'exécution (indices de tableaux, par exemple) et à optimiser le code produit...

Les méthodes existantes cherchent à définir différents domaines en autorisant des contraintes plus ou moins complexes. Dans ce stage, nous avons voulu explorer une méthode orthogonale pour ajuster la précision et la complexité de l'analyse statique. On a cherché à regrouper des sous-ensembles de variables du programme pour analyser finement leurs propriétés. Ces sous-ensembles peuvent être non-disjoints pour permettre l'expression implicite de relations (certes moins précises) entre variables de groupes différents. Ce partitionnement de variables utilise le produit réduit de Cousot. La difficulté est la réduction des ensembles de contraintes d'un tel domaine. Dans ce stage, le produit réduit a été appliqué aux octogones [Min01]. Ce nouveau domaine réduit la complexité des octogones si on l'utilise comme un produit réduit et il généralise également les octogones eux-mêmes (avec la même complexité asymptotique) si l'on ne prend qu'un seul groupe de toutes les variables.

Ce travail repose sur des prérequis qu'il nous faut d'abord rappeler. Le chapitre 1 a pour but de fixer les notions et notations relatives aux programmes que nous considérerons simplement comme des automates interprétés. Le chapitre 2 est un rappel des principes de l'interprétation abstraite, illustré d'un exemple simple d'analyse, suivi des notions nécessaires au produit de domaine. Nous ferons ensuite une présentation rapide des domaines abstraits les plus connus pour l'analyse de propriétés numériques dans le chapitre 3. Le chapitre 4 expose l'intuition des recherches menées ici pour permettre de mieux ajuster la précision de l'analyse. Nous présenterons les points essentiels de l'analyse statique avec les octogones [Min01] dans le chapitre 5 pour introduire le treillis des octogones creux (chapitre 6) qui généralise le produit réduit d'octogones et les octogones eux-mêmes. L'implémentation de ce nouveau domaine est décrite au chapitre 7.

Chapitre 1

Principe d'analyse d'accessibilité

Dans ce chapitre, nous introduisons les principes et définitions nécessaires à l'analyse statique de programme. Pour pouvoir parler de propriétés mathématiques sur un programme, il nous faut définir son sens précis, en donnant un modèle de programme. Ensuite, nous montrerons comment formuler l'analyse d'invariant en se basant sur ce modèle.

1.1 Modèle de programme et Système de transition

Nous nous restreindrons à des programmes sans appel de procédure, sans allocation dynamique, ne manipulant que des variables numériques. Ces caractéristiques soulèvent des problèmes que nous n'avons pas voulu traiter ici. Un programme contient alors un nombre fixe N de variables de type réel.

Remarque 1 *Les programmes avec des variables booléennes peuvent tout de même être analysés en traitant les conditions portant sur des booléens comme des choix indéterministes. Les programmes avec des variables dynamiques peuvent aussi être analysés mais sans retenir d'information sur ces variables.*

Tous les programmes de la forme ainsi définie peuvent être représentés avec des automates interprétés définis comme suit :

Définition 1 (Automate interprété) *Un automate interprété est un triplet* $(K, \xrightarrow{c,a}, K_{init})$*, où*

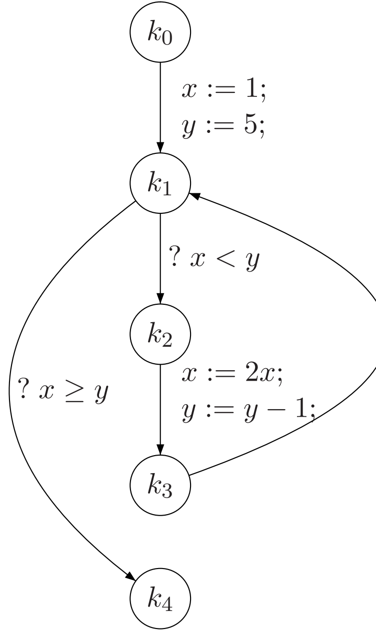
- K *est un ensemble de* points de contrôle*,*
- $\xrightarrow{c,a}$ *est un ensemble de* transitions *définies de la manière suivante :*
  Un élément de $\xrightarrow{c,a}$ *est un quadruplet* (k_1, c, a, k_2) *où*
  - k_1 *est le point de contrôle d'*entrée *de la transition,*
  - $c : \mathbb{R}^N \to \mathbb{B}$*, une* condition *sur les variables à valider pour passer la transition,*
  - $a : \mathbb{R}^N \to \mathbb{R}^N$*, correspond aux* opérations *effectuées sur les variables de la transition,*
  - k_2 *est le point de contrôle de* sortie *de la transition*
- K_{init} *est un sous-ensemble de* K *définissant les différents points d'entrée du programme.*

Exemple 1

Un petit programme

$x := 1$;
$y := 5$;
while $x < y$ ***do***
  $x := 2 \times x$;
  $y := y - 1$;
end while

Automate interprété associé :



Définition 2 (Etat du programme) *Soit K l'ensemble des points de contrôle et $\vec{x} \in V$ une valuation des variables du programme (ici, $V = \mathbb{R}^N$), un état est un couple $(k, \vec{x}) \in K \times V$.*

Définition 3 (Sémantique opérationnelle d'un automate interprété) *La sémantique opérationnelle d'un automate interprété est un système de transition entre états, noté $(S, \rightarrow, S_{init})$ où*

- *S est l'ensemble des états du programme,*
- *$S_{init} = \{(k, \vec{v}) | k \in K_{init}\} \subseteq S$ est l'ensemble des états initiaux,*
- *$\rightarrow$ est la relation de transition entre états.*

Une transition $(k_1, \vec{v}_1) \rightarrow (k_2, \vec{v}_2)$ est définie à partir d'une transition $(k_1, c, a, k_2) \in \xrightarrow{c,a}$ de l'automate interprété :

$$\frac{(k_1, c, a, k_2) \in \xrightarrow{c,a}\ ,\ c(\vec{v}_1) = vrai\ ,\ \vec{v}_2 = a(\vec{v}_1)}{(k_1, \vec{v}_1) \rightarrow (k_2, \vec{v}_2)}$$

Intuitivement, si le programme est dans l'état $(k_1, \vec{x})$, il peut passer dans l'état $(k_2, a(\vec{x}))$ si'l existe une transition (k_1, c, a, k_2) tel que $c(\vec{x})$ est vrai.

En prenant toutes les valuations possibles pour les variables au niveau des états initiaux, nous supposons que les variables ne sont pas initialisées. Cela ne restreint pas l'ensemble des programmes pouvant être analysé, une première transition pouvant être ajoutée pour effectuer cette initialisation.

1.2 Propriété d'accessibilité

Les propriétés sur les variables numériques de nous allons chercher à valider ici sont des propriétés d'accessibilité. Ce n'est pas le cas de toutes les propriétés. Par exemple, les propriétés de sûreté (quelquechose de mauvais n'arrive jamais), de vivacité (une action se produira dans un temps fini) requièrent d'autres types d'analyse.

Nous pouvons définir l'accessibilité d'un état et les propriétés d'accessibilité de la manière suivante :

Définition 4 (Accessibilité d'un état) *Un état est accessible ssi*

1. *il est initial ou,*
2. *il est l'image d'un état initial par une séquence de transitions.*

Définition 5 (Propriétés d'accessibilité) *Soit* $A = (K, \xrightarrow{c,a}, K_{init})$ *un automate interprété et* $Acc(A) \subseteq S$ *l'ensemble des états accessibles. Une propriété d'accessibilité* P *est définie par un ensemble* $S_p \subseteq S$ *d'états.* A *satisfait* P *ssi* $S_p \subseteq Acc(A)$*.*

On note X_k l'ensemble des valuations possibles des variables au point k du programme. Les X_k sont reliés par la sémantique du programme. Avec la définition d'accessibilité d'un état, nous pouvons définir récursivement l'ensemble des états accessibles du programme comme étant les états initiaux et les états images d'un état accessible par une transition. Ces dépendances peuvent être formalisées avec le système d'équations suivant :

$$X_k = X_k^{init} \cup \bigcup_{(k',a,c,k) \in \rightarrow} a(X_{k'} \cap c) \tag{1.1}$$

avec $\forall k \in K,\ X_k^{init} = \begin{cases} \mathbb{R}^N & \text{si k est un état initial,} \\ \emptyset^N & \text{sinon} \end{cases}$

Pour ce système, on étend les fonctions c et a de l'automate interprété de type $\mathbb{R}^N \rightarrow \mathbb{R}^N$ à des fonctions de type $\wp(\mathbb{R}^N) \rightarrow \wp(\mathbb{R}^N)$ qui restreignent les valuations $\vec{x}$ aux conditions et aux opérations qu'elles contiennent :

- c est étendue à une fonction $c : \wp(\mathbb{R}^N) \rightarrow \wp(\mathbb{R}^N)$ où $c(X_k) = \{\vec{x} \mid \vec{x} \in X_k \wedge c(\vec{x}) = vrai\}$
- a est étendue à une fonction $a : \wp(\mathbb{R}^N) \rightarrow \wp(\mathbb{R}^N)$ où $a(X_k) = \{\vec{x} \mid \exists \vec{x'} \in X_k,\ a(\vec{x'}) = \vec{x}\}$

De manière générale, on formulera ce système d'équations $X_k = F_k(X_1, \ldots, X_{|K|})$ pour tout $k \in K$, également noté $X = F(X)$ avec $X = \{X_1, \ldots, X_{|K|}\}$ et $F = (F_1 \ldots F_{|K|})$.

1.3 Résolution de l'équation de point fixe

Le domaine numérique dans lequel est effectué le calcul de point fixe est un treillis complet $(D, \subseteq)$. Sous ces hypothèses, le théorème de Tarski nous assure l'existence d'un plus petit point fixe pour (X_k^n).

Théorème 1 (Théorème de Tarski)
Soit $L = (D, \subseteq)$ *un treillis complet et* $F : D \rightarrow D$ *une fonction monotone,*
alors F *possède un plus petit point fixe* $lfp(F) = \cap\{\vec{x} \in D | F(\vec{x}) \subseteq \vec{x}\}$

Maintenant que l'on sait que ce point fixe existe, il nous faut le calculer. Le théorème de Kleene nous donne une vision constructive de la caractérisation précédente.

Théorème 2 (Théorème de Kleene)
Soit $L = (\wp(C), \subseteq, \cap, \cup, \bot, \top)$ *un treillis complet et* $F : C \rightarrow C$ *une fonction continue,*
alors $lfp(F) = \bigcup_{n \geq 0} F^n(\bot)$ *et*
$gfp(F) = \bigcap_{n \geq 0} F^n(\top)$
avec $F^0(\vec{x}) = \vec{x}$ *et* $F^{n+1}(\vec{x}) = F(F(\vec{x}))$*.*

Remarque 2 $F^0(\bot) = \bot \subseteq F^1(\bot)$ *et comme* F *est croissante, on a* $\forall n \geq 0,\ F^n(\bot) \subseteq F^{n+1}(\bot)$*. La suite* $(x_n) = F^n(\bot)$ *est donc croissante.*

L'ensemble des solutions de ce système d'équations peut alors être obtenu en calculant itérativement les termes de la suite (X^n) définie de la manière suivante :

$$X^0 = \bot \text{ et } X^{n+1} = F(X^n)$$

Si on applique le théorème à l'équation (1.1), on obtient :

$$X_k^0 = X_k^{init} \text{ et } X_k^{n+1} = X_k^{init} \cup \bigcup_{(k',a,c,k) \in \rightarrow} a(X_{k'}^n \cap c)$$

La résolution d'une telle équation pose deux problèmes qui sont d'une part la complexité du domaine de calcul $\wp(\mathbb{R}^N)$ et d'autre part la résolution itérative de l'équation de point fixe. Le domaine concret $\wp(\mathbb{R}^N)$ est trop complexe pour que ces éléments soient représentables de façon efficace et normalisée. Les opérations de ce domaine et sa relation d'ordre doivent pouvoir être calculées. Pour l'équation de point fixe, le calcul itératif de la suite (X^n) peut ne pas terminer quand la hauteur du treillis est infinie, ce qui est très souvent le cas du domaine concret. Nous allons devoir nous autoriser des approximations pour rendre ce problème décidable.

Chapitre 2

Interprétation abstraite

Comme nous l'avons montré au chapitre précédent, la validation de propriétés sur les variables numériques se ramène à un problème d'accessibilité des états du programme. Ce calcul d'accessibilité est équivalent à la resolution d'une équation de point fixe $X = F(X)$, $X \in \wp(\mathbb{R}^N)$ qui est indécidable. L'interprétation abstraite [CC77, CC92b] propose une solution générale pour résoudre le problème de la complexité du domaine et de résolution itérative de l'équation de point fixe. Elle consiste en deux approximations successives. D'une part nous pouvons nous placer dans un domaine plus simple afin de représenter et de manipuler les éléments de ce domaine. Ensuite nous effectuons une approximation dynamique pour la terminaison de l'analyse, due au calcul de point fixe.

2.1 Approximation statique

De manière générale le domaine concret est complexe et nous avons besoin de représenter ses éléments de manière normalisée pour pouvoir les manipuler algorithmiquement. Pour résoudre ce problème de complexité, nous pouvons associer un domaine abstrait plus simple au domaine concret, à condition que ces deux domaines soit reliés par une connexion de Galois.

Définition 6 (Connexion de Galois) *Soient $C(\sqsubseteq)$ et $A(\sqsubseteq^\#)$ 2 ensembles partiellement ordonnés, et des fonctions $\alpha : C \rightarrow A$ et $\gamma : A \rightarrow C$. (C, α, γ, A) forment une connexion de Galois si,*

$$\forall x \in C, \forall y \in A,\ \alpha(x) \sqsubseteq^\# y \Longleftrightarrow x \sqsubseteq \gamma(y)$$

α est la fonction d'abstraction, qui associe à une valeur concrète son approximation dans le domaine abstrait A, tandis que γ est la fonction de concrétisation qui associe à une valeur abstraite sa signification concrète. Ce mécanisme se généralise aux opérateurs :

Définition 7 *Soit (C, α, γ, A) une connexion de Galois et $f : C^n \rightarrow C$ une fonction monotone. On appelle plus petite approximation supérieure de f dans A la fonction :*

$$\begin{array}{llll} \alpha(f) : & A^n & \rightarrow & A \\ & (x_1, \ldots, x_n) & \mapsto & \alpha(f(\gamma(x_1), \ldots, \gamma(x_n))) \end{array}$$

Théorème 3 *Soit (C, α, γ, A) une connexion de Galois et $F : C \rightarrow C$. Si C est un treillis complet et F est monotone, alors $\alpha(lfp(F)) \sqsubseteq^\# lfp(\alpha(F))$*

Ce dernier résultat très important nous assure que nous obtenons toujours une solution surapproximant la plus petite solution de $X = F(X)$ en résolvant l'équation de point fixe $Y = \alpha(F)(Y)$ avec la fonction $\alpha(F)$ dans le domaine abstrait A.

2.2 Approximation dynamique

Après cette première approximation, la limite de la suite (X^n) définie par $X^0 = \bot^\#$ et $X^{(n+1)} = F^\#(X^n)$ peut ne pas converger dans A si la hauteur du treillis est infinie. Une solution à ce problème est d'introduire un opérateur d'élargissement (*widening* en anglais) au niveau des dépendances cycliques entre variables X_k [CC92b]. Il doit permettre de calculer une approximation du calcul de point fixe en un nombre fini de pas.

Définition 8 (Opérateur d'élargissement) *Une fonction $\nabla : A \times A \rightarrow A$ est un opérateur d'élargissement si :*

1. $\forall y_1, y_2 \in A, (y_1 \sqcup^\# y_2) \sqsubseteq^\# (y_1 \nabla y_2)$
2. *Pour toute suite croissante (x_n), la suite croissante définie par $y_0 = x_0,\ y_{n+1} = y_n \nabla x_{n+1}$ converge en un nombre fini d'itérations.*

Théorème 4 *Soit A un treillis complet, $F^\# : A \rightarrow A$ une fonction continue, et $\nabla : A \times A \rightarrow A$ un opérateur d'élargissement. La suite $y_0 = \bot,\ y_{n+1} = (y_n)\nabla F^\#(y_n)$ converge en un nombre fini d'étapes vers un post point fixe y de $F^\#$.*

Une conséquence immédiate nous donne $\mathrm{lfp}(F^\#) \sqsubseteq y$. Cet opérateur nous permet donc d'obtenir une approximation du résultat de l'équation $X = F(X)$ en calculant la limite de la suite (y_n) définie par $y_0 = \bot^\#$ et $y_{(n+1)} = (y_n)\nabla F^\#(y_n)$.

Pour l'intuition, dans les domaines abstraits pour les variables numériques, l'opérateur d'élargissement élimine généralement les contraintes qui sont modifiées à chaque pas.

Il existe de nombreuses améliorations possibles de l'opérateur d'élargissement toujours dans les domaines abstraits :

- On ne peut utiliser l'opérateur d'élargissement qu'après un certain nombre de pas. Généralement, davantage d'invariants sont préservés par l'approximation.
- Le point fixe y obtenu avec cet opérateur peut aussi être amélioré en calculant les premiers termes de la suite descendante $z_0 = y,\ z_{n+1} = G(z_n)$ dans une séquence descendante qui donne toujours une approximation supérieure au plus petit point fixe.
- L'opérateur d'élargissement 'up to' [HPR97] élargit tout en conservant un ensemble de contraintes extrait du programme et n'utilise l'élargissement classique qu'à l'itération suivante. Cela permet souvent d'obtenir un point fixe tout en préservant davantage de contraintes numériques, notamment les contraintes induites des boucles.

2.3 Exemple : L'analyse d'intervalle

2.3.1 Définition

L'analyse d'intervalles [CC76] est le plus ancien domaine abstrait utilisé en interprétation abstraite. Ce domaine associe à un élément de $\wp(\mathbb{R})$ le plus petit intervalle qui le contient. Il est défini par le treillis I où :

$$I = \{\bot\} \cup \{[a,b] \mid a \in \mathbb{R} \cup \{-\infty\} \wedge b \in \mathbb{R} \cup \{+\infty\} \wedge a \leq b\}$$

Les fonctions d'abstraction et de concrétisation se définissent de la manière suivante :

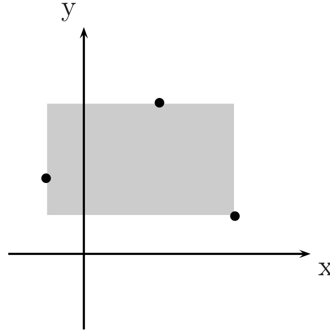


FIG. 2.1 – Intervalles

Abstraction	Concrétisation
$\alpha(\emptyset) = \perp^{\#}$	$\gamma(\perp^{\#}) = \emptyset$
$\alpha(x) = [\inf(x), \sup(x)]$	$\gamma([a,b]) = \{x \mid a \leq x \leq b\}$

Par exemple, cette analyse approxime $\{0, 3, 10\}$ par l'intervalle $[0, 10]$. Ce qui nous donne $\gamma(\alpha(\{0, 3, 10\})) = \{x \mid 0 \leq x \leq 10\}$. La généralisation pour analyser plusieurs variables se fait en abstrayant $\wp(\mathbb{R}^N)$ par I^N.

Les union, intersection et opération d'élargissement du treillis des intervalles sont définis comme suit :

$$\text{Union} \begin{cases} A \cup^{\#} B = A \ \text{ si } B = \perp \\ \qquad\quad\ = B \ \text{ si } A = \perp \\ [a,b] \cup^{\#} [a',b'] = [\min(a,b), \max(a',b')] \ \text{ sinon} \end{cases}$$

$$\text{Intersection} \begin{cases} A \cap^{\#} B = \perp \ \text{ si } A \text{ ou } B = \emptyset \\ [a,b] \cap^{\#} [a',b'] = \emptyset \ \text{ si } a > b' \text{ ou } a' > b \\ \qquad\qquad\qquad = [\max(a,a'), \min(b,b')] \ \text{ sinon} \end{cases}$$

$$\text{Opérateur d'élargissement} \begin{cases} \perp \nabla [a',b'] \quad = [a',b'] \\ [a,b] \nabla [a',b'] = [c,d] \text{ avec } c = -\infty \ \text{ si } a' < a, \ a \text{ sinon et,} \\ \qquad\qquad\qquad\qquad\qquad d = +\infty \ \text{ si } b' > b, \ b \text{ sinon.} \end{cases}$$

L'opérateur ∇ des intervalles éliminent toutes les bornes qui évoluent à chaque pas de l'analyse. Par exemple, si x est un compteur initialisé à 0 dans une boucle , on évaluera $[0,0]\nabla[0,1] = [0, +\infty]$, ce qui a bien pour effet de faire terminer le calcul de point fixe.

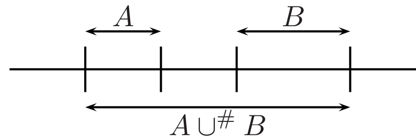


FIG. 2.2 – Union d'intervalles

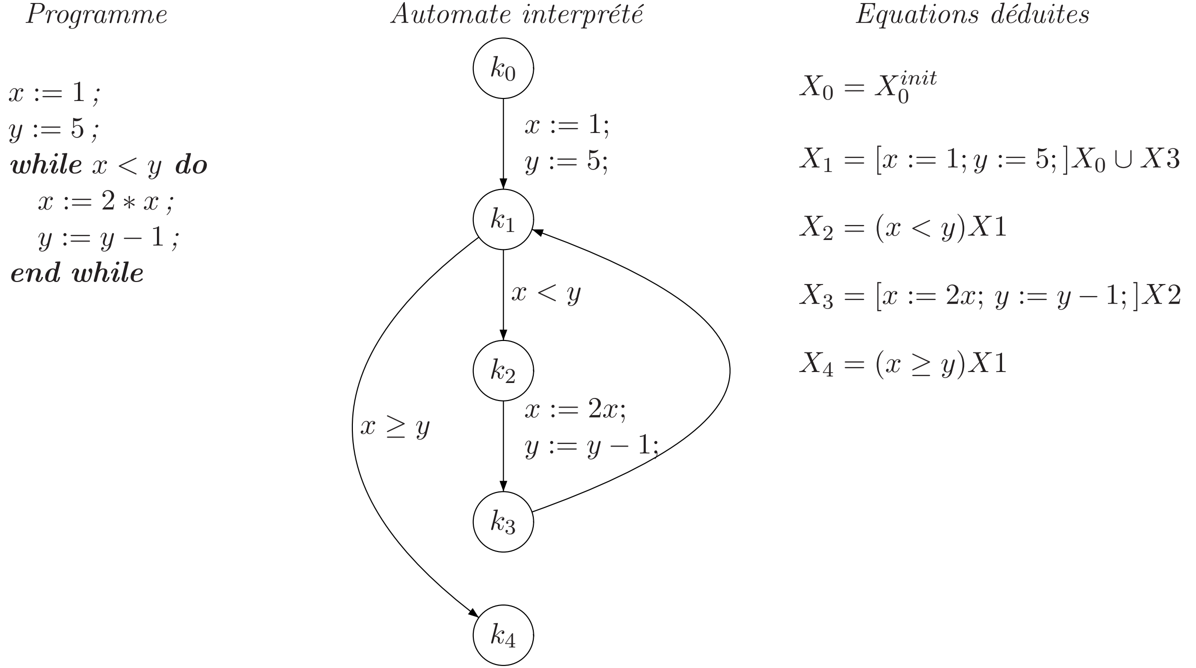
Remarque 3 *Comme le montre la figure 2.2, dans le cas où* $(a > b' \vee a' > b)$*, l'union contient plus que l'union ensembliste entre 2 intervalles pour ne conserver qu'un seul intervalle par variable.*

2.3.2 Exemple d'analyse d'intervalle sur un programme

Afin d'illustrer toutes les notions précédentes nous allons appliquer une analyse d'intervalles sur le programme déjà utilisé dans la première partie.

On note $[a](c)X_k$ l'image $X_{k'}$ de la valuation X_k par la transition (k, c, a, k').

Exemple 2



Calcul itératif des X_k , $k \in \{0 \dots 4\}$ avec abstraction des opérateurs concrets jusqu'à obtention d'un point fixe (l'intervalle de gauche représente x, celui de droite y) sans utilisation de l'opération d'élargissement :

pas	0	1	2	3	4	5	6
X_0	$\bot$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$
X_1	$\bot$	[1,1],[5,5]	[1,2],[4,5]	[1,4],[3,5]	[1,8],[2,5]	[1,8],[1,5]	[1,8],[1,5]
X_2	$\bot$	[1,1],[5,5]	[1,2],[4,5]	[1,4],[3,5]	[1,4],[2,5]	[1,4],[2,5]	[1,4],[2,5]
X_3	$\bot$	[2,2],[4,4]	[2,4],[3,4]	[2,8],[2,4]	[2,8],[1,4]	[2,8],[1,4]	[2,8],[1,4]
X_4	$\bot$	$\bot$	$\bot$	[3,4],[3,4]	[2,8],[2,5]	[1,8],[1,5]	[1,8],[1,5]

Au pas 7 nous obtenons les mêmes solutions qu'au pas précédent. Un point fixe est donc atteint ; nous obtenons des bornes sup et inf pour les variables (x et y) à chaque point du programme.

Remarque 4 *Nous obtenons le même résultat en 3 pas en utilisant l'opération d'élargissement au* 2e *pas et une séquence descendante au* 3e *:*

pas	1	2	3
X_0	$\top$	$\top$	$\top$
X_1	[1,1],[5,5]	[1,+∞],[−∞,5]	[1,8],[1,5]
X_2	[1,1],[5,5]	[1,4],[2,5]	[1,4],[2,5]
X_3	[2,2],[4,4]	[2,8],[1,4]	[2,8],[1,4]
X_4	$\bot$	[1,+∞],[−∞,5]	[1,8],[0,5]

2.4 Construction de domaines abstraits

2.4.1 Réduction et forme canonique

Il peut exister plusieurs représentations abstraites d'un élément concret. Par exemple, $\{x, y \mid x = 1 \wedge y = x + 3\}$ et $\{x, y \mid x = 1 \wedge y = 4\}$ représentent le même ensemble. Pour beaucoup d'opérations au cours de l'analyse, on a besoin d'une représentation abstraite unique d'un même élément concret (test d'égalité, inclusion ...). Pour cela, nous allons définir une relation d'équivalence :

Définition 9 *Soit une connexion de Galois (C, α, γ, A). Deux éléments abstraits a_1 et a_2 sont* équivalents *ssi* $\gamma(a_1) = \gamma(a_2)$

La forme normale des éléments d'une même classe d'équivalence sera généralement le plus petit élément selon la relation d'ordre abstraite ($\sqsubseteq^\#$). Il arrive, dans certains domaines abstraits, que le plus petit élément ne soit pas unique. Des choix arbitraires sont alors nécessaires. L'opération associant à un élément sa forme canonique s'appelle la réduction. C'est souvent l'opération la plus complexe algorithmiquement.

2.4.2 Produit

Une analyse peut reposer sur plusieurs domaines abstraits pour conjuguer l'expressivité de chacun. On appelle produit l'utilisation simultanée de plusieurs domaines. On forme un nouveau domaine A avec le produit d'un ensemble A_i $i \in I$ de domaines abstraits. Cela se formalise de la manière suivante :

Définition 10 (Produit) *Le produit des connexions de Galois $(C, \alpha_i, \gamma_i, A_i)$, $i \in \{1, 2\}$ partageant le même domaine concret est la connexion (C, α, γ, A) où :*

1. *$A = A_1 \times A_2$, ordonné par l'ordre partiel $\sqsubseteq$ suivant : $(a_1, a_2) \sqsubseteq (a'_1, a'_2) \Leftrightarrow a_1 \sqsubseteq a'_1 \wedge a_2 \sqsubseteq a'_2$*
2. *$\alpha(c) = (\alpha_1(c), \alpha_2(c)$*
3. *$\gamma(a_1, a_2) = \gamma(a_1) \sqcap \gamma(a_2)$*

Produit réduit Un produit réduit [CC92a] est simplement une réduction sur un produit de domaines. Il se définit facilement mais il est souvent compliqué à mettre en œuvre. Par exemple, une analyse d'un produit d'intervalles et de parité peut donner $\{1 \leq x \leq 4\} \wedge \{x \in \text{Impair}\}$ qui doit se réduire en $\{1 \leq x \leq 3\} \wedge \{x \in \text{Impair}\}$, ou encore $\{x = 3\} \wedge \{x \in \text{Pair}\}$ doit être détecté insatisfiable. La difficulté vient du fait que l'intersection de deux éléments concrets $(\gamma(a_1) \sqcap \gamma(a_2))$ n'est pas calculable algorithmiquement. On est donc ramené à propager les contraintes d'un domaine à un autre.

Chapitre 3

Exemples de domaines abstraits pour les variables numériques

Pour avoir un aperçu de l'état de l'art en ce qui concerne les domaines numériques abstraits, nous présentons brièvement les plus connus. Le domaine des intervalles présenté dans le chapitre précédent est très efficace mais également assez peu expressif. En effet, il ne peut exprimer aucune relation entre variables (ex : $v_i \leq v_j$). Nous allons présenter les treillis de la famille des polyèdres (zones, octogones, 2 variables par équation, polyèdres, template), plus précis et plus complexes que les intervalles. Nous conclurons sur une comparaison et une discussion sur ces différents domaines avant d'exposer la problématique abordée dans ce rapport.

On appelle *opération* dans un domaine abstrait n'importe quelle transformation nécessaire à l'analyse (union, intersection, projection, abstraction de condition ou d'affectation ...). Par habitude, la complexité des domaines est évaluée par rapport à la complexité de ces opérations. N est le nombre de variables $Var = \{v_1, \dots v_N\}$ du programme.

3.1 Zones et Octogones

Zones Le treillis des zones exprime des bornes sur toutes les variables ($a \leq v \leq b$) et différences de variables ($a \leq v_i - v_j \leq b$) [Yov98].

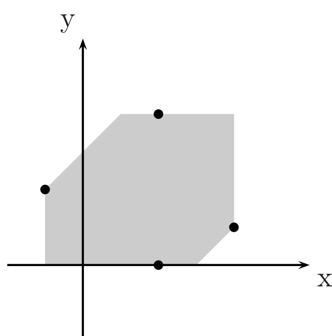


Fig. 3.1 – Zones

Ce type de contraintes est également appelé contrainte de potentiels [CLR94]. Les ensembles de contraintes de potentiels sont classiquement représentés par des graphes. Les opérations sont quasi directement des algorithmes de graphe (détection d'un circuit de poids négatif et calcul

de plus court chemin). Grace aux algorithmes utilisés, les zones héritent d'une complexité de $O(N^3)$ pour ces opérations.

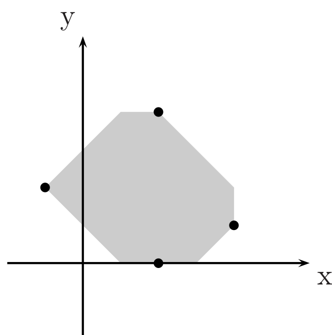


FIG. 3.2 – Octogones

Octogones Le treillis des octogones [Mine01] étend le précédent. On peut ici représenter toutes les contraintes de la forme $(av_i + bv_j \leq c)$, avec $a, b \in \{-1, 0, +1\}$. L'ensemble des contraintes est représenté par un graphe de $2N$ sommets, ce qui donne une complexité spatiale en $O(N^2)$. Cette technique de représentation s'appuie sur les zones et contient toutes les sommes et les différences entre chaque couple de variables. Les opérations utilisent des algorithmes classiques de graphe, adaptés au domaine, sans augmenter leur complexité $O(N^3)$ en temps. C'est un compromis entre une complexité cubique et un treillis relationnel assez expressif. Nous reviendrons plus en détail sur ce treillis dans la cinquième partie.

3.2 2 variables par équation

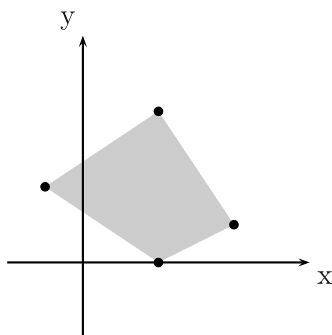


FIG. 3.3 – 2 variables par équation

Ce treillis [SKH02] généralise le précédent en contenant toutes les inéquations sur deux variables : $av_i + bv_j \leq c$ avec $a, b, c \in \mathbb{R}$, les coefficients des variables peuvent maintenant être dans $\mathbb{R}$ au lieu de $\{-1, 0, +1\}$ comme les octogones. Ce domaine très expressif peut inférer de nouvelles inéquations à chaque point de contrôle du programme. En revanche, le nombre d'inéquations n'est plus borné, la complexité spatiale est en $O(p)$ avec p inéquations. Ce treillis permet une meilleur précision de l'analyse au prix d'un coût mémoire non borné. Pour une meilleur efficacité, les opérations nécessaires à l'analyse ne travaillent jamais dans $\mathbb{R}^N$, on réalise d'abord une projection dans le plan (ou l'espace) des 2 (ou 3) variables utilisées pour effectuer les calculs dans $\mathbb{R}^2$ ou $\mathbb{R}^3$. Les opérations ne dépassent alors jamais la complexité en $O(N^2.p.log(p))$

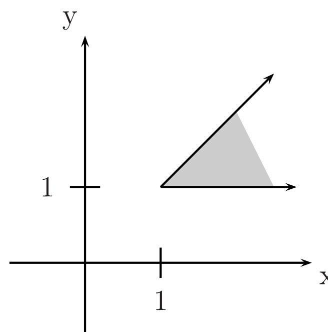
en temps, avec N variables et p inéquations. Ce résultat est intéressant comparé aux octogones, mais il est quand même supérieur en complexité.

3.3 Polyèdres convexes

Ce treillis [CH78, HPR97] est capable de représenter toutes les conjonctions de contraintes linéaires sur toutes les variables du programme : $a_1v_1 + \ldots + a_Nv_N \leq c$ où $v_1, \ldots, v_N$ sont les variables du programme, $a_1, \ldots, a_N$ des coefficients rationnels, et c une constante réelle. Ce domaine plus expressif que le précédent peut également inférer de nouvelles inéquations à chaque point de contrôle du programme. Le nombre d'inéquations n'est toujours pas borné (la représentation géométrique reste la même que pour le treillis 2 var. par équations en deux dimensions). Il existe deux représentations duales d'un même polyèdre.

Le polyèdre ci-contre peut être défini comme :

- 2 inéquations : $y \geq 1$ et $y \leq x$ ou bien
- un ensemble de points : (1, 1) et
  un ensemble de rayons (demi-droites) : (1,1) et (1,0).



Ces deux représentations sont nécessaires, car les différentes opérations s'effectuent alternativement dans l'une ou l'autre représentation et le passage d'une représentation à l'autre est responsable de la complexité temporelle $O(p.2^N)$, (p : nombre d'inéquations ou de générateurs, N : nombre de variables). Sa complexité spatiale $O(p.N)$ est due au stockage des p inéquations contenant N coefficients.

Ce treillis est capable de prouver des propriétés très intéressantes que ce soit pour des systèmes réactifs, des problèmes d'exclusion ... [HPR97] Cependant, sa forte complexité permettra difficilement le passage à l'échelle de cette analyse si le nombre de variables est élevé.

3.4 Les templates

Cette façon, très récemment proposée, de représenter les contraintes [SSM05] est un peu différente des précédentes. Au lieu de restreindre le treillis des polyèdres par le nombre de variables ou les coefficients d'une inéquation du type des polyèdres ($a_1v_1 + \ldots + a_Nv_N \leq c$), on va fixer le nombre d'inéquations, et définir à l'avance leurs formes en fixant la valeur des coefficients $a_1, \ldots, a_N$. Une contrainte peut avoir la même forme que dans le treillis des polyèdres, mais une fois l'ensemble des inéquations fixé, seules les bornes c des inéquations pourront être calculées. Selon le choix des inéquations, ce treillis peut généraliser les intervalles, les zones et les octogones puisque le nombre de contraintes est borné pour ces treillis.

Les opérations se font en utilisant la programmation linéaire sur les contraintes, elles sont alors de complexité temporelle $O(p.N^k)$ où p est le nombre d'inéquations, N^k est le coût de la programmation linéaire. La complexité spatiale est la même que celle du treillis des polyèdres $O(p.N)$. Cependant, le nombre p d'inéquations est ici fixé par l'utilisateur, et donc borné. La complexité est contrôlée, et dépend totalement de ce nombre. La précision de l'analyse dépend aussi complètement du choix des inéquations. Un choix aléatoire ne donnerait vraisemblablement aucun résultat. Plusieurs méthodes pour effectuer ce choix existent, et une amélioration possible

de ce treillis (nous ne le détaillerons pas ici) est d'associer un ensemble différent d'inéquations par point de contrôle du programme.

3.5 Discussion

Notion de treillis relationnel Certains treillis présentés ont la particularité d'être relationnels. Un treillis A est relationnel si l'abstraction du domaine concret en un domaine $A_{x,y}$ (domaine sur les variables x et y) peut apporter plus d'informations que la conjonction des domaines A_x et A_y. Si tel est le cas, on peut en conclure que le treillis exprime des relations entre x et y. Tous les treillis présentés ici sont relationnels à l'exception des intervalles. Par exemple, ils peuvent exprimer $x = y$ ou $x > y + 10$.

Nous avons vu précédemment que nous avons le choix de décider quel treillis utiliser pour effectuer une analyse statique. Ce choix est déterminant pour la précision et le temps de l'analyse. Si le treillis des *intervalles* est simple en complexité (linéaire en fonction du nombre de variables), il est aussi assez faible en précision. Il peut être utilisé pour éviter les erreurs du type overflows, mais ne s'avère pas assez précis pour beaucoup d'autres propriétés.

Le treillis des *polyèdres* permet une analyse de précision mais au prix d'une complexité spatiale non bornée et d'une complexité exponentielle des opérations abstraites en fonction du nombre de variables.

Le treillis des *2 variables par équation* restreint les polyèdres aux inéquations ne contenant que deux variables $av_i + bv_j \leq c$. Tout en pouvant encore exprimer des contraintes assez variées, il réduit la complexité des opérations. Il est lui même supérieur en précision aux treillis des *octogones*, [Min01] qui permet la même forme d'inéquations mais seulement avec des coefficients a et $b \in \{-1, 0, 1\}$. De nouveau, la complexité et la précision de l'analyse baissent.

On peut aussi restreindre les polyèdres en fixant à l'avance le nombre d'inéquations et leurs formes (treillis *template* [SSM05]). A chaque point de contrôle est associé un certain nombre d'inéquations de la forme $a_1v_1 + \ldots + a_Nv_N \leq c$ avec des coefficients fixés à l'avance. Ce choix est bien sûr à faire avec attention, en tenant compte de critères venant du programme.

Les complexités maximales des opérations nécessaires à l'analyse statique des différents treillis présentés est donné par le tableau suivant :

	Intervalle	octogone	2 var/éq.$^{(1)}$	Template$^{(2)}$	Polyèdre$^{(1)}$
Temps	$O(N)$	$O(N^3)$	$O(N^2.p.\log(p))$	$O(p.N^k)$	$O(p.2^N)$
Place	$O(N)$	$O(N^2)$	$O(p)$	$O(p.N)$	$O(p.N)$

(N : nombre de variables, p : nombre d'inéquations, N^k : complexité de la programmation linéaire, $^{(1)}$: p est non borné, $^{(2)}$: p est borné fixé par l'utilisateur)

Fig. 3.4 – Complexités temporelles et spatiales des opérations

Il s'agit de la complexité théorique dans le cas le pire qui est parfois bien supérieure à la complexité pratique. Une précision doit également être apportée afin de ne pas faire d'erreurs dans la comparaison entre les Templates et les polyèdres ou les 2 variables par équation. Pour les templates, le nombre p d'inéquations est fixé lors de l'analyse et donc borné, contrairement aux 2 autres treillis dont la complexité est également fonction de p, où p n'est pas borné.

Les Templates peuvent être utilisés pour instancier les treillis des intervalles, des octogones ou des octaèdres ; sachant que les contraintes de ces treillis sont représentées par un nombre borné d'inéquations, il suffit de les instancier dans un treillis Template pour obtenir la même précision

que ces treillis. Cependant, d'un point de vue algorithmique, les octogones et les octaèdres sont fondés sur des techniques plus efficaces. On obtient un classement des treillis présentés en fonction de leur précision en terme de propriété :

- Polyèdre > 2 variables par équation > octogone > zone > intervalle,
- Polyèdre > Templates.

Chapitre 4

Problématique du stage

Le but de ce stage est de permettre d'ajuster la précision (et la complexité) de l'analyse grâce à des techniques différentes que le choix présenté dans le chapitre précédent. L'idée ici est de grouper les variables du programme par paquets, puis de moduler la précision des contraintes en fonction de l'appartenance des variables en question à un même groupe ou non. Cela s'exprime grâce au produit réduit de domaines introduit au chapitre 2. Nous avons choisi le treillis des octogones pour appliquer ce 'partitionnement'.

4.1 Principe

La solution pour réduire la complexité de l'analyse présentée ici, se fonde sur l'observation de dépendances plus ou moins fortes entre les variables. On propose de grouper par paquets les variables ayant de forte relations entre elles.

Soit Var l'ensemble des variables du programmes à analyser, et $Var_1 \cup \ldots \cup Var_n = Var$ un ensemble de paquets de variables, avec $|Var| = N$ et $|Var_i| = N_i$. On note $(\wp(\mathbb{R}^N), \alpha_V, \gamma_V, A[V])$ la connexion de Galois où $A[V]$ est un domaine abstrait décrivant les propriétés sur les variables V.

La connexion de Galois $(\wp(\mathbb{R}^N), \alpha, \gamma, A[Var])$ entre le domaine concret $\wp(\mathbb{R}^N)$ et un treillis de dimension $|Var|$ est la solution proposée dans la plupart des analyses statiques. Soient

$$(\wp(\mathbb{R}^N), \alpha_i, \gamma_i, A[Var_i])$$

les n connexions de Galois entre le même domaine concret $\wp(\mathbb{R}^N)$ et n domaines abstraits de dimension $|Var_i|$. Ici, la solution orthogonale au choix du treillis lui-même est d'utiliser ces n domaines abstraits pour en constituer un seul : le produit des n domaines abstraits $(A[Var_1] \times \ldots \times A[Var_n])$.

4.2 Exemples

Prenons un ensemble de 4 variables $\{w, x, y, z\}$ et utilisons le treillis des octogones. La technique classique consiste à utiliser un seul domaine $Oct_{w,x,y,z}$ pour représenter les valeurs des 4 variables. Une autre possibilité est d'abstraire les variables par paquets en prenant par exemple $Oct_{w,x} \times Oct_{y,z}$. Cette technique réduit la complexité de cette analyse de $\mathrm{O}(N^3)$ en $\mathrm{O}(2 \times (N/2)^3)$ où $N = 4$. Elle a aussi l'inconvénient de ne pouvoir représenter aucune relation entre $\{w, x\}$ et $\{y, z\}$.

Cette idée a déjà été expérimentée par l'équipe Cousot [BCC+03] où ils proposent un outil Astrée donnant de bons résultats qui s'intéresse à vérifier des erreurs à l'éxecution du type Run

Times Error. Dans nos recherches, nous nous sommes interressé à des propriétés de plus haut niveau. De plus, ils n'effectuent pas de réduction sur le produit de domaines, mais seulement sur chacun des domaines séparément. De ce fait $\{0 < w < y\} \times \{y < z < 0\}$ n'est pas détecté insatisfiable. Nous proposons ici un produit réduit de domaines, ce qui signifie que tous les systèmes insatisfiables sont détectés.

Une solution, intermédiaire entre un seul ensemble de variables et un partitionnement, est de partager des variables entre les paquets. Reprenons l'ensemble de 4 variables $\{w, x, y, z\}$, nous pouvons regrouper les variables en $Oct_{w,x,y} \times Oct_{x,y,z}$. Le gain de complexité est un peu diminué, mais l'expressivité de ce domaine est beaucoup plus forte. En effet, il permet de représenter implicitement des relations entre les variables de paquets différents par l'intermédiaire des variables x et y redondantes (ex : $\{w < y\} \times \{y < z\} \Rightarrow (w < z)$).

4.3 Produit réduit d'octogones

Dans ce stage, on s'est intéressé à appliquer cette idée et ce produit réduit de domaines A_i aux octogones. Ce cas est plus simple que le produit réduit de polyèdres parce que le treillis est lui-même plus simple. Le nombre de contraintes pouvant être inféré à borné $\{\pm v_i \pm v_j \leq c \mid \forall v_i, v_j \in Var\}$ où Var est l'ensemble fini des variables du programme.

Considérer le produit réduit comme plusieurs domaines interagissant les uns avec les autres est une vision difficile. Nous allons adopter une représentation bien plus simple et générale du produit réduit d'octogones. Nous pouvons exploiter le fait que le nombre de contraintes est borné en considérant le produit d'octogone comme un seul octogone ne s'intérressant qu'à un sous-ensemble de contraintes de $\{\pm v_i \pm v_j \leq c \mid \forall v_i, v_j \in Var\}$.

Par exemple, le produit $Oct_{x,y} \times Oct_{y,z}$ peut être vu comme un octogone $Oct_{x,y,z}$ où on oublie volontairement les contraintes entre x et z. Nous appellons ce type d'octogones *les octogones creux*.

Ce treillis des octogones creux généralise le produit réduit d'octogones. En effet il suffit d'y intégrer toutes les relations exprimées dans chaque octogone pour obtenir la même expressivité. Il généralise également les octogones eux-mêmes car on peut aussi représenter la totalité des contraintes d'un octogone dans un octogone creux.

4.4 Choix des paquets

Le choix des paquets influence bien sûr beaucoup fondamental pour la précision de ce treilis. Ce problème n'a pas été traité dans le cadre de ce stage. C'est un problème qui a été rencontré pour le treillis des Templates présenté dans le chapitre 3 et par l'équipe Cousot dans leur outil.

Chapitre 5

Le treillis des Octogones en détail

Pour expliquer le treillis des octogones creux [Min01, Min04], nous allons tout d'abord revenir sur le treillis des octogones plus en détail pour expliquer quelles sont les difficultés dues à la représentation des contraintes ($\pm v_i \pm v_j \leq c$) et enfin quelles sont les opérations les plus complexes algorithmiquement.

5.1 Les zones

Les octogones se sont très fortement inspirés des zones qui expriment une conjonction de contraintes sur les différences de variables $\{v_i - v_j \leq c_{ij} \mid v_i, v_j \in Var\}$.

$$Z = \{\perp\} \cup \{c_{i,j} \mid 1 \leq i, j \leq N \wedge c_{ij} \in \mathbb{R} \cup \{+\infty\} \wedge c_{ij} \geq -c_{ji}\}$$

Une zone utilise un graphe pour stocker la valeur des bornes c_{ij} de chaque différence de variables. Aux sommets sont associés les variables, aux poids des arcs les bornes c. Le poids de l'arc (v_i, v_j) est la valeur de la borne c_{ij} de l'inéquation $(v_i - v_j \leq c_{ij})$. Un sommet $\mathbf{0}$ est ajouté afin de permettre la représentation des intervalles. De cette manière, les contraintes $v_i \leq c_{i\mathbf{0}}$ (resp. $v_i \leq c_{\mathbf{0}i}$) sont représentées par les arcs $(v_i, \mathbf{0})$ (resp. $(\mathbf{0}, v_i)$).

Définition 11 (Représentation d'une zone) *Une zone sur N variables est représentée par un graphe complet $G = \{Q, E, w\}$ avec :*

- *Q : ensemble de sommets, $Q = Var \cup \{\mathbf{0}\}$, chaque sommet $q_i \neq \mathbf{0}$ représente la variable v_i, $\mathbf{0}$ représente la valeur 0,*
- *$E = Q \times Q$: ensemble d'arrêtes, $|E| = (|Var| + 1)^2$,*
- *$w : E \rightarrow \mathbb{R}$*

Définition 12 (Sémantique d'une zone) *Soit une zone représentée par un graphe $G = \{Q, E, w\}$, alors*

$$\gamma(G) = \{\vec{v} = (v_1, \ldots, v_N) \mid (q_i, q_j) \in E \Rightarrow v_i - v_j \leq w(q_i, q_j)\}$$

Cette représentation a été choisie pour ses propriétés intéressantes. Les opérations nécessaires à l'analyse trouve des correspondances avec des algorithmes de graphes classiques . On prouve que le système de contraintes est satisfiable si et seulement si il existe un circuit de poids négatif dans le graphe. Si le système de contraintes est satisfiable, l'élément canonique équivalent est obtenu grâce à un calcul de plus court chemin entre chaque couple de sommets (intuition Fig. 5.1). Ainsi, grâce aux algorithmes classiques sur les graphes, on obtient des complexités raisonnables, $O(N^3)$ au pire en temps pour les opérations.

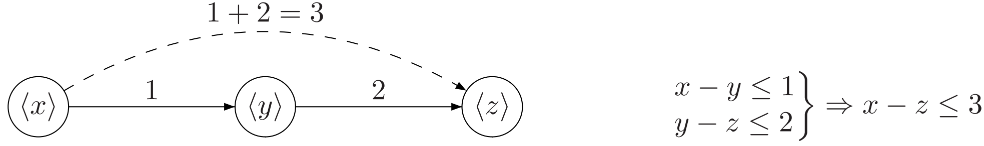


FIG. 5.1 – Exemple de minimisation d'une contrainte dans une zone

Théorème 5 (Satisfiabilité d'un ensemble de contraintes de potentiels)
Etant donné un système d'inéquations composées d'une différence sur 2 variables, soit G le graphe correspondant. Si G ne contient aucun circuit de poids négatif, alors il existe une solution réalisable pour le système.

Preuve Voir [CLR94], §25.5, Introduction to Algorithms. □

5.2 Représentation d'un octogone

Pour pouvoir représenter toutes les contraintes de la forme $(av_1 + bv_2 \leq c)$ avec $a, b \in \{-1, 0, 1\}$, Antoine Miné [Min01] duplique la représentation des variables dans le graphe. A une variable v sont associés deux sommets $\langle -1, v\rangle$ et $\langle +1, v\rangle$ représentant intuitivement $-v$ et $+v$. Le poids d'un arc représente maintenant une différence de variables multipliées par -1 ou $+1$. L'arc $(\langle s_i, v_i\rangle, \langle s_j, v_j\rangle)$ représente la contrainte $s_i v_i - s_j v_j \leq c$, où c est le poids de cet arc. La représentation des octogones n'inclut pas le sommet $\mathbf{0}$.

Définition 13 (Représentation d'un octogone) *Un octagone sur N variables est représenté par un graphe complet $G = \{Q, E, w\}$ avec :*

- *Q : ensemble des sommets du graphe, $Q = \{\langle s, v\rangle | \forall v \in Var, \forall s \in \{-1, 1\}\}$, $|Q| = 2N$. Q contient $\langle -1, v\rangle$ et $\langle +1, v\rangle$ pour chaque variable $v \in Var$,*
- *E : $E = Q \times Q$ ensemble des arrêtes du graphe $|E| = 4N^2$*
- *$w : E \to \mathbb{R}$*

Définition 14 (Sémantique d'un octogone) *Soit un octogone représenté par un graphe $G = \{Q, E, w\}$, alors*

$$\gamma(G) = \{\vec{v} = (v_1, \ldots, v_N) \mid (\langle s_i, v_i\rangle, \langle s_j, v_j\rangle) \in E \Rightarrow s_i v_i - s_j v_j \leq w(\langle s_i, v_i\rangle, \langle s_j, v_j\rangle)\}$$

Cette représentation permet la représentation de toutes les contraintes $(av_i + bv_j \leq c)$ avec $a, b \in \{-1, 0, 1\}$ et aussi des contraintes $(\pm v \leq c)$ grâce aux arcs $(\langle 1, v\rangle, \langle -1, v\rangle)$ pour $v \leq c$ et $(\langle -1, v\rangle, \langle 1, v\rangle)$ pour $-v \leq c$. Cela permet au treillis des octogones dinclure le treillis des intervalles. Notons que le poids de ces arcs est égal au double de ces bornes.

Toutes les contraintes sont dupliquées avec cette représentation (excepté celles des intervalles). En effet, si $v_i \neq v_j$, les deux arcs $(\langle s_i, v_i\rangle, \langle s_j, v_j\rangle)$ et $(\langle -s_j, v_j\rangle, \langle -s_i, v_i\rangle)$ représente la même inéquation $(s_i v_i - s_j v_j \leq c)$. Cette redondance n'est pas suprimée car nécessaire au bon fonctionnement des algorithmes utilisés.

Cohérence On dira qu'un octogone est cohérent si $w(\langle s_i, v_i\rangle, \langle s_j, v_j\rangle) = w(\langle -s_j, v_j\rangle, \langle -s_i, v_i\rangle)$ pour tous arcs du graphe.

5.3 Clôture et forme canonique

L'opération la plus importante et la plus difficile à réaliser est l'opération de réduction. Elle est composée du test de vacuité (vérifier que l'ensemble des contraintes est satisfiable), puis de la minimisation des contraintes. Cette opération est par exemple nécessaire après chaque ajout de contrainte ou chaque intersection. En cas de satisfiabilité, la minimisation des contraintes est aussi indispensable car elle découvre de nouvelles contraintes (par exemple $(x - y \leq 2 \wedge x + y \leq -2) \Rightarrow x \leq 0$). Sans l'expression explicite de toutes les contraintes représentables, on peut ne pas détecter l'inaccessibilité d'un état.

Comme les zones, cette opération s'aide d'algorithmes de graphe (Fig. 5.2). C'est l'algorithme de Floyd-Warshall de calcul de plus court chemin qui est utilisé ici.

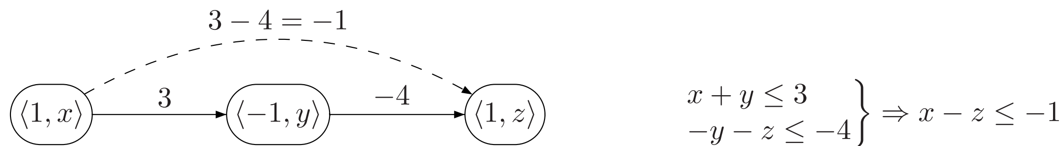


Fig. 5.2 – Exemple de minimisation d'une contrainte dans un octogone

Un problème de taille vient de la combinaison des contraintes $(\pm v_i \pm v_j \leq c)$ et $(\pm v \leq c)$. La difficulté ici vient du fait que les arcs $(\langle s_i, v_i\rangle, \langle -s_i, v_i\rangle)$ et $(\langle s_j, v_j\rangle, \langle -s_j, v_j\rangle)$ de poids c_i et c_j signifient $s_i v_i \leq c_i/2$ et $s_j v_j \leq c_j/2$ mais un algorithme de plus court chemin ne permet pas de déduire $s_i v_i + s_j v_j \leq c_i/2 + c_j/2$. Dans les octogones, ce problème est résolu en maintenant cette relation à la main à chaque itération de l'algorithme. Ces modifications n'augmentent pas la complexité temporelle ($\mathrm{O}(N^3)$) mais augmente considérablement la taille de la preuve (6 pages pour cette opération).

Chapitre 6

Le treillis des Octogones creux

Nous présentons une adaptation du treillis des octogones expliqué dans le chapitre précédent. Le treillis des octogones creux sert à effectuer le produit réduit d'octogones grâce à sa particularité de pouvoir contenir n'importe quel sous-ensemble de contraintes représentable par un octogone classique.

6.1 Représentation

Pour le treillis des octogones, la représentation utilisée est un graphe complet. Ici, pour les sparse-octagons, nous utilisons aussi un graphe mais qui ne sera pas complet et on lui réintroduit le sommet **0** comme dans les zones. Sa représentation et sa sémantique ressemblent à celles des octogones :

Définition 15 (Représentation d'un octogone creux) *Un octogone creux sur N variables est représenté par un graphe $G = \{Q, E, w\}$ avec :*

- *Q : ensemble des sommets du graphe, $Q = \{\langle 0, 0\rangle\} \cup \{\langle s, v\rangle | v \in Var,\ s \in \{-1, 1\}\}$, $|Q| = 2N + 1$. Q contient $\langle -1, v\rangle$ et $\langle +1, v\rangle$ pour chaque variable x , plus un sommet $\langle 0, 0\rangle$ qui représente une variable **0** non signée pour permettre la représentation des intervalles $\pm v \leq c$.*
- *E : $E \subseteq Q \times Q$ ensemble des arrêtes du graphe*
- *$w : E \to \mathbb{R}$*

Définition 16 (Sémantique d'un octogone creux) *Soit un octogone creux représenté par un graphe $G = \{Q, E, w\}$, alors*

$$\gamma(G) = \{\vec{v} = (v_1, \ldots, v_N) \mid (\langle s_i, v_i\rangle, \langle s_j, v_j\rangle) \in E \Rightarrow s_i v_i - s_j v_j \leq w(\langle s_i, v_i\rangle, \langle s_j, v_j\rangle)\}$$

La fonction de concrétisation γ associe à un graphe G l'ensemble de toutes les valuations possibles des N variables qui respectent toutes les inéquations du graphe.

Comme pour les octogones, toutes les contraintes sont dupliquées (exceptées celles des intervalles). Les deux arcs $(\langle s_i, v_i\rangle, \langle s_j, v_j\rangle)$ et $(\langle -s_j, v_j\rangle, \langle -s_i, v_i\rangle)$ représentent la même inéquation $(s_i v_i - s_j v_j \leq c)$ si $v_i \neq v_j$.

Pour simplifier la notation, on notera q^- le sommet $\langle -s, v\rangle$ si q est le sommet $\langle s, v\rangle$, et **0** le sommet $\langle 0, 0\rangle$. On utilisera la fonction l pour dénoter de la longueur du plus court chemin entre deux sommets.

6.2 Opération de réduction

Cette opération décrite de manière générale dans le chapitre 2, doit rendre unique la représentation abstraite des éléments appartenant à la même classe d'équivalence (Définition 9). On choisit parmi ces éléments le plus petit.

L'opération de réduction va donc chercher à minimiser la valeur des contraintes, c'est à dire le poids des arcs du graphe, de façon optimale. La réduction se fait grâce aux calculs des plus courts chemins entre chaque couple de sommets. L'algorithme de Floyd-Warshall utilisé pour les octogones est adapté pour les graphes denses. Ici, nous utilisons l'algorithme de Johnson adapté au graphe peu dense. Il calcule les plus courts chemins entre chaque couple de sommets en $O(N^2.\log(N) + N|E|)$ avec N nombre de sommets et $|E|$ nombre d'arcs. L'algorithme de Johnson est modifié pour pouvoir servir d'algorithme de réduction.

La réduction effectue tout d'abord un test de vacuité, afin de savoir si l'élément G est devenu vide, puis une minimisation des contraintes nécessitant une repondération positive du poids des arcs du graphe. Nous allons présenter ces différentes étapes.

6.2.1 Test de vacuité

La réduction d'un octogone creux commence par un test de satisfiabilité. Ce test s'effectue en détectant si le graphe possède un circuit de poids négatif. Systématiquement, un circuit de poids négatif (ex : $w(a,b) = 2$; $w(b,c) = -4$; $w(c,a) = 1$;) peut se traduire par une contrainte insatisfiable :

$$\left.\begin{array}{l} a - b \leq 3 \\ b - c \leq -4 \\ c - a \leq 1 \end{array}\right\} \Rightarrow 0 \leq -1$$

Théorème 6 (Satisfiabilité des contraintes d'un sparse-octagon) .
Soit G un graphe cohérent,

$$\gamma(G) = \emptyset \Leftrightarrow \exists \text{ un circuit de poids strictement négatif}$$

Preuve

- $\Leftarrow$:
  Supposons $\gamma(G) \neq \emptyset$, et démontrons que l'existence d'un circuit de poids négatif amène à une contradiction. Soit $p = (q_1, \ldots, q_n = q_1)$ un circuit de poids négatif dans G. Soit $\vec{v} \in \gamma(G)$, alors $\forall 1 \leq i \leq n-1$, $\vec{c}(q_i, q_{i+1}).\vec{v} \leq w(q_i, q_{i+1})$. Donc

  $$\sum_{i=1}^{n-1} \vec{c}(q_i, q_{i+1})\vec{v} \leq \sum_{i=1}^{n-1} w(q_i, q_{i+1}) < 0$$

  Or $\sum_{i=1}^{n-1} \vec{c}(q_i, q_{i+1}) = 0$. On en déduit que $0 < 0$. C'est une contradiction, donc $\gamma(G) \neq \emptyset \Rightarrow \nexists$ un circuit de poids strictement négatif.

- $\Rightarrow$:
  Nous allons démontrer la contraposée,

  $$\nexists \text{ un circuit de poids strictement négatif } \Rightarrow \gamma(G) \neq \emptyset$$

  D'après le théorème 5, s'il n'existe pas de circuit de poids négatif, alors $\gamma^{Zone}(S) \neq \emptyset$. On va montrer maintenant que $\gamma^{Zone}(S) \neq \emptyset \Rightarrow \gamma(S) \neq \emptyset$.

Soit $\vec{x} = (x_0, x_1, \ldots, x_{2N}) \in \gamma^{Zone}(S)$, une valuation ordonnée des $2N+1$ variables $\mathbf{0}, \langle 1, v_1\rangle, \langle -1, v_1\rangle, \ldots, \langle 1, v_N\rangle, \langle -1, v_N\rangle$. Cette valuation $\vec{v}$ respecte les contraintes $\forall (q_i, q_j) \in E, x_i - x_j \leq w(q_i, q_j)$. On a également l'impliquation suivante :

$$x_i - x_j \leq c \Rightarrow \forall a \in \mathbb{R},\ (x_i - a) - (x_j - a) \leq c$$

En prenant x_0 à la place de a dans l'expression précédente, on en déduit que si $\vec{x} = (x_0, x_1, \ldots, x_{2N}) \in \gamma^{Zone}(S)$ alors $(x_0 - x_0 = 0, x_1 - x_0, x_2 - x_0, \ldots, x_{2N} - x_0) \in \gamma^{Zone}(S)$. Soit $\vec{y}$ cette nouvelle valuation dont la coordonnée y_0 associée au sommet $\mathbf{0}$ respecte $y_0 = 0$.

On note y_i^- la coordonnée y_{i-1} si i est pair et y_{i+1} si i est impair. Par hypothèse, on a : $\forall (q_i, q_j) \in E, y_i - y_j \leq w(q_i, q_j)$ et $y_j^- - y_i^- \leq w(q_j^-, q_i^-)$. Par cohérence du graphe, on a : $w(q_i, q_j) = w(q_j^-, q_i^-)$, donc si $\vec{y} = (0, y_1, \ldots, y_{2N}) \in \gamma^{Zone}(S)$ alors $\vec{y'} = (0, y_1^-, \ldots, y_{2N}^-) = (0, y_2, y_1, y_4, y_3, \ldots, y_{2N}, y_{2N-1}) \in \gamma^{Zone}(S)$ également. Comme $\gamma^{Zone}(S)$ est une intersection de demi-espaces, sa représentation est convexe, et donc $\vec{z} = (\vec{y} + \vec{y'})/2 \in \gamma^{Zone}(S)$. On a :

$$\begin{cases} z_0 = 0 \\ \forall i \in \{1..N\},\ z_{2i} = (y_{2i} + y'_{2i})/2 = (-y'_{2i-1} - y_{2i-1})/2 = -z_{2i-1} \end{cases}$$

ce qui signifie que $\forall z$ la valuation d'un sommet q, la valuation z^- du sommet q^- respecte $z^- = -z$. Donc $(z_1, z_3, \ldots, z_{2N-1}) \in \gamma(S)$ et $\gamma(S) \neq 0$.

□

L'algorithme de Bellman-Ford utilisé pour détecter s'il existe un circuit de poids négatif fonctionne de la manière suivante :

- en entrée : le graphe et un sommet initial v
- en sortie : un booléen affecté à faux, si le graphe contient un circuit de poids négatif
  un booléen affecté à vrai et les plus courts chemins du sommet initial v à tous les autres sinon

La liste l_v retournée sert de base de calcul pour une repondération du graphe afin de pouvoir exécuter l'algorithme de Dijkstra.

6.2.2 Minimisation des contraintes : propriétés utilisées

Si l'élément n'a pas été détecté vide, on doit maintenant minimiser ses contraintes. Pour cela nous calculons pour tous couples (a, b) de sommets le poids minimal des chemins partant de a et aboutissant à b. Un premier calcul de plus court chemin doit être effectué pour minimiser les contraintes des intervalles, un second minimise toutes les autres contraintes.

Ces deux étapes sont nécessaires car nous avons ici une difficulté ressemblant à celle rencontrée dans les octogones. Typiquement, la valeur d'un chemin de $(\langle +1, v\rangle)$ à $(\langle -1, v\rangle)$ représentant la contrainte $(v \leq c)$ n'est pas utilisé par un algorithme de plus court chemin pour minimiser les arcs $(\langle +1, v\rangle, \mathbf{0})$ et $(\mathbf{0}, (\langle -1, v\rangle))$ représentant la même contrainte.

Notation On note $\vec{c}(q)$ la valeur $(\overbrace{0, \ldots, 0, \underbrace{s}_{i}, 0, \ldots, 0}^{N fois})$ et $\vec{c}(\mathbf{0})$ le vecteur nul $(\overbrace{0, \ldots, 0}^{N fois})$

Remarque 5 *Sémantiquement, à chaque chemin $p = (q_1, \ldots, q_n)$ correspond une inéquation :*

$$\sum_{i=1}^{n-1} \vec{c}(q_i, q_{i+1}) \cdot \vec{v} \leq \sum_{i=1}^{n-1} w(q_i, q_{i+1})$$

Algorithm 1 Algorithme de Bellman-Ford $(G, r) \rightarrow (cycle, l_r)$

$l_r(q) := +\infty;\ \forall q \in Q$
$l_r(r) := 0;$
for all $k = 0$ to N **do**
  for all $(q_i, q_j) \in E$ **do**
    if $l_r(q_j) > l_r(q_i) + w(q_i, q_j)$ **then**
      $l_r(q_j) := l_r(q_j) + w(q_i, q_j);$
    end if
    // invariant : $\forall q \in Q,\ l_r(q) =$ *valeur minimale des chemins* r *à* q *de longueur* $\leq k$
  end for
end for
for all $(q_i, q_j) \in E$ **do**
  if $l_r(q_j) > l_r(q_i) + w(q_i, q_j)$ **then**
    Return (false,nil) ;
  end if
end for
Return (true,l_r) ;

qui peut se réduire à

$$\vec{c}(q_1) \cdot \vec{v} - \vec{c}(q_n) \cdot \vec{v} \leq \sum_{i=1}^{n-1} w(q_i, q_{i+1})$$

car $\sum_{i=1}^{n-1} \vec{c}(q_i, q_{i+1}) = \vec{c}(q_1) - \vec{c}(q_n)$. *Dans le cas d'un circuit* $p = (q_1, \ldots, q_n = q_1)$ *et l'inéquation devient* $\sum_{i=1}^{n-1} w(q_i, q_{i+1}) \geq 0$

Rappel : on note $l(q_1, q_2)$ le poids du plus court chemin entre les sommets q_1 et q_2.

Lemme 1 *Dans un graphe* G *cohérent,* $\forall q, r \in Q$,

1. $l(q, r) = l(r^-, q^-)$ *et* $l(q, \mathbf{0}) = l(\mathbf{0}, q^-)$
2. $l(q, q^-) \leq 2l(q, \mathbf{0})$
3. *Si* $p = (q_1 = q, \ldots, q_n, \mathbf{0}, q'_1, \ldots, q'_m = q^-)$ *est un plus court chemin, alors* $\exists p'$ *un plus court chemin entre* q *et* q^- *pouvant s'écrire* $(q, \ldots, q_n, \mathbf{0}, q_n^-, \ldots, q^-)$ *et* $l(q, q^-) = 2l(q, \mathbf{0})$

Preuve

1. Par cohérence du graphe, à tout chemin $(q_1 = q, q_2, \ldots, q_{n-1}, q_n = r)$, on peut associer un chemin de même poids entre r^- et q^- s'écrivant $(q_n^- = r^-, q_{n-1}^-, \ldots, q_2^-, q_1^- = q^-)$, ce qui implique que $\forall q, r \in Q,\ l(q, r) = l(r^-, q^-)$ et comme $\mathbf{0} = \mathbf{0}^-$, on a aussi $l(q, \mathbf{0}) = l(\mathbf{0}, q^-)$.
2. Par inégalité triangulaire, $l(q, q^-) \leq l(q, \mathbf{0}) + l(\mathbf{0}, q^-)$. Comme $l(q, \mathbf{0}) = l(\mathbf{0}, q^-)$, on a $\forall q \in Q,\ l(q, q^-) \leq 2l(q, \mathbf{0})$
3. Soit $p = (q_1 = q, \ldots, q_n, \mathbf{0}, q'_1, \ldots, q'_m = q^-)$ un plus court chemin entre q et q^-. Alors le sous-chemin de p $(q_1, \ldots, q_n, \mathbf{0})$ est aussi un plus court chemin. De plus, $l(q, \mathbf{0}) = l(\mathbf{0}, q^-)$ donc le sous-chemin de p du sommet $\mathbf{0}$ au sommet q^- peut se réécrire $(\mathbf{0}, q_n^-, \ldots, q_2^-, q_1^- = q^-)$. On en conclue que $p' = (q_1 = q, \ldots, q_n, \mathbf{0}, q_n^-, \ldots, q_1^- = q^-)$ est un plus court chemin et que $l(q, q^-) = l(q, \mathbf{0}) + l(\mathbf{0}, q^-) = 2l(q, \mathbf{0})$.

□

Le lemme suivant nous assure que le calcul des contraintes ($v \leq c$) peut se faire itérativement sans interagir entre elles. G' est le graphe où les contraintes supportées par les arcs $(\mathbf{0}, q_1^-)$ et $(q_1, \mathbf{0})$ sont minimisées.

Lemme 2 (Invariance des plus courts chemins de q à q^-)
Soit 2 graphes cohérents $G = \{Q, E, w\}$ et $G' = \{Q, E, w'\}$ où

$$\begin{aligned} w'(e) &= l_G(q_1, q_1^-)/2 \text{ si } e \in \{(\mathbf{0}, q_1^-), (q_1, \mathbf{0})\} \\ &= w(e) \text{ sinon,} \end{aligned}$$

alors $\forall q \in Q, l_{G'}(q, q^-) = l_G(q, q^-)$

Preuve Soit G et G', 2 graphes définis comme ci-dessus. Par définition, $w'(q_1, \mathbf{0}) = l_G(q_1, q_1^-)/2$ et d'après le lemme 1, $l_G(q_1, q_1^-)/2 \leq l_G(q_1, \mathbf{0})$ donc $w'(q_1, \mathbf{0}) \leq w(q_1, \mathbf{0})$. On en déduit que $\forall e \in E,\ w'(e) \leq w(e)$, et donc que

$$\forall q \in Q, l_{G'}(q, q^-) \leq l_G(q, q^-)$$

Nous allons maintenant supposer que $l_{G'}(q, q^-) < l_G(q, q^-)$ et démontrer que cela amène à une contradiction.

Si les plus courts chemins dans G' de q à q^- ne passent pas par les arcs $(q_1, \mathbf{0})$ et $(q_1, \mathbf{0})$, alors ils ne passent que par des arcs dont le poids est égal à ceux dans G et $l_{G'}(q, q^-) = l_G(q, q^-)$.

Maintenant prenons le cas où un plus court chemin dans G' passe par $(q_1, \mathbf{0})$ et/ou $(q_1, \mathbf{0})$. Par cohérence du graphe, ces deux possibilités se démontrent de manière similaire. Si ce chemin passe par $(q_1, \mathbf{0})$, il peut s'écrire $(q, \ldots, q_1, \mathbf{0}, \ldots, q^-)$.

D'après le lemme 1 (3.), $p = (q, \ldots, q_1, \mathbf{0}, q_1^-, \ldots, q^-)$ est aussi un plus court chemin dans G'. Tous les poids des arcs de ce chemin sont égaux à ceux du graphe G exceptés $(q_1, \mathbf{0})$ et $(\mathbf{0}, q_1^-)$. Et comme on a supposé $l_{G'}(q, q^-) < l_G(q, q^-)$, on doit avoir

$$w'(q_1, \mathbf{0}) + w'(\mathbf{0}, q_1^-) < l_G(q_1, q_1^-)$$

D'autre part, on a $w'(q_1, \mathbf{0}) = w'(\mathbf{0}, q_1^-) = l_G(q_1, q_1^-)/2$ par hypothèse, donc

$$w'(q_1, \mathbf{0}) + w'(\mathbf{0}, q_1^-) = l_G(q_1, q_1^-)$$

C'est une contradiction, on en conclut que $\forall q \in Q,\ l_G(q, q^-) = l_{G'}(q, q^-)$. □

Le calcul des plus courts chemins entre q et q^- pour tout $q \in Q$ peut donc se faire successivement en modifiant le graphe. Ces calculs n'interfèrent pas entre eux. Nous formalisons maintenant la notion de contraintes minimales.

Définition 17 (Saturation d'une contrainte) *Une contrainte représentée par un arc $e \in E$ est saturée si l'une des deux conditions est satisfaite :*

1. $w(e) < +\infty$ *et* $\exists \vec{x} \in \gamma(S)$ *tel que* $\vec{c}(e) \cdot \vec{x} = w(e)$
2. $w(e) = +\infty$ *et* $\forall a < +\infty,\ \exists \vec{x} \in \gamma(S)$ *tel que* $\vec{c}(e) \cdot \vec{x} \geq a$

Si la contrainte $(\langle s_y, y\rangle, \langle s_z, z\rangle)$ est saturée alors, soit son poids est $+\infty$ et cela signifie qu'on peut trouver une concrétisation dont la valeur de $s_y y - s_z z$ soit supérieure à n'importe quel réel, soit son poids est réel et on doit pouvoir trouver un élément de $\gamma(G)$ où la valuation de y et z vérifie $s_y y - s_z z = w(\langle s_y, y\rangle, \langle s_z, z\rangle)$, c'est à dire aucune plus petite borne ne pourra être considérée sans modifier l'ensemble $\gamma(G)$.

Le théorème suivant nous prouve les contraintes ($v \leq c$) sont saturées après avoir calculé les plus courts chemins entre les sommets q et q^- et après avoir modifié les poids des arcs $(q, \mathbf{0})$ et $(\mathbf{0}, q^-)$.

Théorème 7 (Saturation des contraintes $\pm x \leq c$) . *Dans un graphe G cohérent où $\forall q \in Q, w(q,\mathbf{0}) = w(\mathbf{0},q^-) = l(q,q^-)/2$, les contraintes $(q,\mathbf{0})$ et $(\mathbf{0},q)$ sont saturées.*

Preuve On va prouver successivement les 2 cas de la définition 17.

1. 1er cas : $w(e) < \infty$
   On va construire un graphe G' tel que $\gamma(G') = \{\vec{x} \mid \vec{x} \in \gamma(G) \wedge \vec{c}(q,\mathbf{0}) \cdot \vec{x} = w(q,\mathbf{0})\}$, et en prouvant ensuite que $\gamma(G') \neq \emptyset$, on aura démontré que la contrainte $(q,\mathbf{0})$ est saturée. Par cohérence du graphe, le cas des contraintes $(\mathbf{0},q)$ se démontre de manière similaire.
   On note $G' = \{Q,E,w'\}$ le graphe égal à $G = \{Q,E,w\}$ où

$$\begin{aligned} w'(e) &= -w(q,\mathbf{0}) \text{ si } e \in \{(q^-,\mathbf{0}),(\mathbf{0},q)\} \\ &= w(e) \text{ sinon,} \end{aligned}$$

   G' est cohérent. La seule contrainte ajoutée dans le graphe G' est $w'(q,\mathbf{0}) \leq \vec{c}(q) \cdot \vec{x}$. Comme G contient déjà $\vec{c}(q) \cdot \vec{x} \leq w'(q,\mathbf{0})$, on obtient le $\gamma(G')$ voulu. On va supposer qu'il existe un circuit de poids négatif dans G' (c'est à dire $\gamma(G') = \emptyset$), et on va montrer que tous les cas mènent à une incohérence.
   - Si le circuit de poids négatif ne passe ni par l'arc $(q^-,\mathbf{0})$ ni par l'arc $(\mathbf{0},q)$, il existerait aussi dans G, donc G serait vide et ne pourrait pas contenir de contrainte bornée.
   - Le circuit de poids négatif passe par un (et un seul) des 2 arcs $(q^-,\mathbf{0})$, $(\mathbf{0},q)$. Par cohérence du graphe, les 2 cas se démontrent de manière similaire. On va prendre le cas où le circuit passe par $(q^-,\mathbf{0})$. Sans perte de généralité, on peut écrire ce circuit $(q_1 = q^-, q_2 = \mathbf{0}, \ldots, q_n = q^-)$. On en déduit que $w'(q^-,\mathbf{0}) + \sum_{k=2}^{n-1} w(q_k,q_{k+1}) < 0$. Comme $w'(q^-,\mathbf{0}) = -w(q,\mathbf{0})$, on peut réécrire l'inégalité en

$$\sum_{k=2}^{n-1} w(q_k,q_{k+1}) < w(q,\mathbf{0})$$

     D'autre part, $w(q,\mathbf{0}) = l_G(q,q^-)/2$, et $l_G(q,q^-)/2 \leq l_G(\mathbf{0},q^-)$ d'après le lemme 1. De plus, par inégalité triangulaire, on a $l_G(\mathbf{0},q^-) \leq \sum_{k=2}^{n-1} w(q_k,q_{k+1})$. On en déduit que

$$w(q,\mathbf{0}) \leq \sum_{k=2}^{n-1} w(q_k,q_{k+1})$$

     Les deux inégalités étant contradictoires, il n'existe pas de circuit négatif.
   - Le circuit de poids négatif passe par les 2 arcs $(q^-,\mathbf{0})$ et $(\mathbf{0},q)$. Comme on peut ne considérer que des circuits simples, sans perte de généralité, on peut écrire ce circuit $(q_1 = q^-, q_2 = \mathbf{0}, q_3 = q, \ldots, q_n = q^-)$. On peut en déduire que $\sum_{k=3}^{n-1} w(q_k,q_{k+1}) + w'(q^-,\mathbf{0}) + w'(\mathbf{0},q) < 0$, et comme $w'(q^-,\mathbf{0}) = w'(\mathbf{0},q) = -w(q,\mathbf{0})$, on peut réécrire l'inégalité en

$$\sum_{k=3}^{n-1} w(q_k,q_{k+1}) < 2w(q,\mathbf{0})$$

     D'autre part $w(q,\mathbf{0}) = l(q,q^-)/2$ par definition, et $l(q,q^-) \leq \sum_{k=3}^{n-1} w(q_k,q_{k+1})$ par inégalité triangulaire, donc

$$w(q,\mathbf{0}) \leq \sum_{k=3}^{n-1} w(q_k,q_{k+1})$$

C'est encore une contradiction, donc il n'existe pas de circuit de poids négatif dans G', donc $\gamma(G') \neq \emptyset$ et la contrainte est saturée.

2. 2nd cas : $w(e) = \infty$
Soit $a \in \mathbb{R}$ et e un arc $(q, \mathbf{0}) \in E$ avec $w(e) = +\infty$, on prend un graphe G' égal à G excepté $w'(\mathbf{0}, q) = w'(q^-, \mathbf{0}) = min(-a)$. On montre de la même manière que précédemment que $\gamma(G') = \{\vec{x} \,|\, \vec{x} \in \gamma(G) \wedge \vec{c}(q, \mathbf{0})\vec{x} > a\} \neq \emptyset$.

□

Saturation de toutes les contraintes

Définition 18 (Saturation d'un octogone creux) .
Un sparse-Octagon représenté par un graphe $G = \{Q, E, w\}$ est saturé si $\forall e \in E$, e est saturée.

Théorème 8 (Caractérisation d'un octogone creux saturé) *Dans un graphe $G = \{Q, E, w\}$ non vide,*

$$\left.\begin{array}{l} \forall q \in Q, w(q, \mathbf{0}) = w(\mathbf{0}, q^-) = l(q, q^-)/2 \;\; et \\ \forall (q, r) \in E, w(e) = l(e) \;\; si \;\; q, r \neq \mathbf{0} \end{array}\right\} \Rightarrow G \text{ est saturé et cohérent.}$$

Preuve Cette preuve est assez similaire à celle du théorème 7. D'après le théorème 7, les contraintes $(q, \mathbf{0})$ et $(\mathbf{0}, q)$ sont saturées $\forall q \in Q$. On va maintenant prouver les 2 cas de la définition 17 pour les contraintes (q, r) où $q, r \neq \mathbf{0}$. Par hypothèse les contraintes $(q, \mathbf{0})$ sont cohérentes avec les contraintes $(\mathbf{0}, q^-)$, et $\forall (q, r) \in E, w(e) = l(e)$ si $q, r \neq \mathbf{0}$ et avec le lemme 1, on obtient que G est cohérent.

1. On va construire un graphe G' tel que $\gamma(G') = \{\vec{x} \mid \vec{x} \in \gamma(G) \wedge \vec{c}(q, r) \cdot \vec{x} = w(q, r)\}$, et en prouvant ensuite que $\gamma(G') \neq \emptyset$, on aura démontré que la contrainte (q, r) est saturée.
On note $G' = \{Q, E, w'\}$ le graphe égal à $G = \{Q, E, w\}$ où $w' = w$ excepté :

$$w'(r, q) = w'(q^-, r^-) = -w(q, r)$$

G' est cohérent. La seule contrainte ajoutée dans le graphe G' est $w'(q, r) \leq \vec{c}(q, r) \cdot \vec{x}$. Comme G contient déjà $\vec{c}(q, r) \cdot \vec{x} \leq w'(q, r)$, on obtient le $\gamma(G')$ voulu. On va supposer qu'il existe un circuit de poids négatif dans G' (c'est à dire $\gamma(G') = \emptyset$), et on va montrer que tous les cas mènent à une incohérence.
– Si le circuit de poids négatif ne passe ni par l'arc (r, q) ni par l'arc (q^-, r^-), il existerait aussi dans G, donc G serait vide.
– Le circuit de poids négatif passe par un (et un seul) des 2 arcs $(r, q), (q^-, r^-)$. Par cohérence du graphe, les 2 cas se démontrent de manière similaire. On va prendre le cas où le circuit passe par (r, q). Sans perte de généralité, on peut écrire ce circuit $(q_1 = r, q_2 = q, \ldots, q_n = r)$. On en déduit que $w'(q, r) + \sum_{k=2}^{n-1} w(q_k, q_{k+1}) < 0$. Comme $w'(r, q) = -w(q, r)$, on peut réécrire l'inégalité en

$$\sum_{k=2}^{n-1} w(q_k, q_{k+1}) < w(q, r)$$

D'autre part, $w(q,r) = l_G(q,r)$, et par inégalité triangulaire $l_G(q,r) \leq \sum_{k=2}^{n-1} w(q_k, q_{k+1})$. On en déduit que

$$w(q,r) \leq \sum_{k=2}^{n-1} w(q_k, q_{k+1})$$

Les deux inégalités étant contradictoires, il n'existe pas de circuit négatif.

– Le circuit de poids négatif passe par les 2 arcs (r,q) et (q^-, r^-). Sans perte de généralité, on peut écrire ce circuit $(q_1 = r^-, \ldots, q_m = r, q_{m+1} = q, \ldots, q_{n-1} = q^-, q_n = r^-)$. Donc :

$$\sum_{k=1}^{m-1} w(q_k, q_{k+1}) + w'(r,q) + \sum_{k=m+1}^{n-2} w(q_k, q_{k+1}) + w'(q^-, r^-) < 0$$

Par définition, $w'(r,q) = w'(q^-, r^-) = -w(q,r)$, donc on peut réécrire l'inégalité en

$$\sum_{k=1}^{m-1} w(q_k, q_{k+1}) + \sum_{k=m+1}^{n-2} w(q_k, q_{k+1}) < 2w(q,r)$$

Par inégalité triangulaire, $l(r^-, r) \leq \sum_{k=1}^{m-1} w(q_k, q_{k+1})$ et $l(q, q^-) \leq \sum_{k=m+1}^{n-2} w(q_k, q_{k+1})$, l'inégalité peut donc se réécrire en

$$l(r^-, r) + l(q, q^-) < 2w(q,r)$$

De plus, par hypothèse, $w(\mathbf{0}, r) = l(r^-, r)/2$, $w(q, \mathbf{0}) = l(q, q^-)/2$ et $w(q,r) = l(q,r)$, ce qui nous donne

$$w(q, \mathbf{0}) + w(\mathbf{0}, r) < l(q,r)$$

D'autre part, $l(q,r) \leq w(q,\mathbf{0}) + w(\mathbf{0}, r)$ par inégalité triangulaire. Ces deux dernières inéquations étant contradictoires, il n'existe pas de circuit négatif.

2. Soit $a \in \mathbb{R}$ et e un arc $(q,r) \in E$ avec $w(e) = +\infty$, on prend un graphe G' égal à G excepté $w'(r,q) = w'(q^-, r^-) = min(w(q,r), -a)$. On montre de la même manière que précédemment que $\gamma(G') = \{\vec{x} \mid \vec{x} \in \gamma(G) \wedge \vec{c}(q,r)\vec{x} > a\} \neq \emptyset$.

□

6.2.3 Algorithme de réduction

L'algorithme de Dijkstra calcule les plus courts chemins d'un sommet vers tous les autres pour des graphes de poids positif. Nous utilisons la liste résultat de Bellman-Ford pour calculer une repondération positive du graphe. Nous avons appelé Dijkstra2 l'algorithme qui effectue le calcul de plus court chemin en repondérant le graphe.

L'algorithme de Johnson clasique calcule les plus courts chemins pour tous couples de sommets en utilisant Dijkstra. Il est modifié pour servir d'algorithme de réduction. On applique une première fois Dijkstra à partir de tous les sommets pour récupérer la valeur minimale des arcs $(q, \mathbf{0})$, puis une deuxième fois pour minimiser la totalité des contraintes. De cette manière, nous obtenons toutes les contraintes minimisées en $O(E.Q.\log(Q))$ si on utilise un tas binaire pour la file de priorité $(Q - L)$, et $O(Q.E + Q^2.\log(Q))$ avec un tas de fibonacci. Les algorithmes de repondération, de Bellman-Ford et de Dijkstra constituent l'algorithme de Johnson modifié que nous utilisons pour réduire un sparse-Octagon.

Algorithm 2 Dijkstra $(G = (Q, E, w), r) \rightarrow (Q \rightarrow \mathbb{R})$

```
// construction d'une fonction l_r, l_r(q) contiendra le poids minimal des chemins de r à q
for all q ∈ Q do
  l_r(q) := +∞ ;
end for
l_r(r) := 0;
L := ∅
while Q − L ≠ ∅ do
  // invariants :∀q ∈ L, l_r(q) est le poids minimal des chemins de r à q.
                 ∀q ∈ (succ(L) − L), l_r(q) est le poids minimal des chemins de r à q
                 passant uniquement par des sommets s_1, ..., s_n ∈ L
                 ∀q ∉ (succ(L) − L), l_r(q) = +∞
  choisir q ∈ (Q − L) tel que l_r(q) soit minimale ;
  L := L ∪ q;
  for all successeur t ∈ Q − L de q do
    l_r(t) := min(l_r(t), l_r(q) + w(q, t));
  end for
end while
return l_r ;
```

Algorithm 3 Repondérer $(G = (Q, E, w)), l_r \rightarrow G$

```
for all (u, v) ∈ E do
  w'(u, v) := w(u, v) + l_r(u) − l_r(v);
end for
Return G ;
```

Algorithm 4 Récupérer $(G = (Q, E, w)), l_r \rightarrow G$

```
for all (u, v) ∈ E do
  w'(u, v) := w(u, v) − l_r(u) + l_r(v);
end for
Return G ;
```

Algorithm 5 Dijkstra2 $(G = (Q, E, w), r, l) \rightarrow (Q \rightarrow \mathbb{R})$

```
G := Repondérer(G, l)
l_r := Dijkstra(G, r)
G := Récupérer(G, l)
return l_r;
```

Algorithm 6 Réduire $(G = (Q, E, w)) \rightarrow G = (Q, E, w)$

// Test de vacuité et repondération du graphe
(cycle,l_i, G) :=Bellman-Ford(G) ;
if (cycle) **then**
  Return $\perp$;
end if
// Dijkstra à partir de chaque sommet pour saturer les contraintes $\pm v \leq c$
// Voir Lemme 2 et Théorème 7
for all $q \in Q$ **do**
  l_q :=Dijkstra2(G, q) ;*
end for
for all $q \in Q - \mathbf{0}$ **do**
  $w(q, \mathbf{0}) := l_q(q^-)/2;$
  $w(\mathbf{0}, q^-) := l_q(q^-)/2;$
end for
// Dijkstra à partir de chaque sommet pour saturer toutes les contraintes
// Voir Théorème 8
for all $q \in Q$ **do**
  l_q :=Dijkstra2(G, q) ; **
end for
for all $(a, b) \in E$ **do**
  $w((a, b)) := l_a(b)$; **
  // calcul des poids non repondérés
  G :=récupérer(G, l_i) ;
end for
Return G ;

* : Cet appel à Dijkstra n'est nécessaire que pour le poids des chemins (q, q^-), donc l'algorithme de Dijkstra peut être modifié pour s'arrêter juste après avoir atteint le sommet q^-) à partir du sommet q.

** : Ces 2 boucles, servant à minimiser toutes les contraintes, peuvent être remplacées par :

```
L := ∅;
for all q ∈ Q do
  // Dijkstra à partir de chaque sommet
  // invariant : ∀q ∈ L, ∀q' ∈ succ(q), w(q,q') = poids minimal des chemins de q à q'
  l_q :=Dijkstra(G, q);
  for all q' ∈ succ(q) do
    w(q,q') := l_q(q');
  end for
  L := L ∪ {q};
end for
```

De cette manière, on affecte les plus courts chemins calculés après chaque exécution de Dijkstra pour éviter d'utiliser une matrice $2N.2N$ en mémoire. Dans ce cas une liste $2N$ suffit.

6.3 Comparatif Octogone/Octogone creux

L'opération de réduction est la plus coûteuse. La réduction d'un octogone s'effectue en $O(N^3)$. Pour le treillis des octogones creux, la réduction s'effectue en $O(N.E + N^2 \log(N))$. Asymptotiquement, pour un graphe dense, les complexités des octogones et des octogones creux sont identiques.

	Octogone	Octogone creux (tas de fibonacci)	Octogone creux (tas binaire)
Complexité temporelle	$O(N^3)$	$O(NE + N^2 \log(N))$	$O(NE \log(N))$
Complexité spatiale	$O(N^2)$	$O(p)$	$O(p)$

Dans le cas où on prend n variables par paquets et où on ne partage aucune variable entre ces paquets, on obtient N/n paquets. Si on utilise les octogones, la complexité est $N/n\times$ $O(n^3)$ soit $O(Nn^2)$. Alors que l'utilisation des octogones creux donne $O(N.(N/n \times n^2) + N^2 \log(N)) = O(N^2 n + N^2 \log(N))$. Notre approche est moins bonne dans ce cas. Une solution a ce problème d'efficacité est peut-être de détecter les composantes fortement connexes du graphe pour pouvoir les traiter séparément.

Prenons maintenant :
- N : nombre de variables dans le programme
- n + k : nombre de variables par paquet
- k : nombre de variables partagées entre deux paquets
- on obtient $\frac{N}{n}$ octogones et $A = \frac{N}{n} \times (n^2 + 2nk)$, c'est à dire $N(n + 2k)$ inéquations représentables

Nous obtenons alors une complexité de $O(N^2(n + 2k) + N^2 \log(N))$ pour une analyse de ce type. Pour le même résultat, les octogones sont obligés de considérer toutes les contraintes et obtiennent une complexité de $O(N^3)$.

Une amélioration non négligeable entre les octogones et les octogones creux est la taille de la preuve. Antoine Miné a besoin d'une dizaine de pages pour prouver son algorithme de réduction. Avec les octogones creux, la preuve est nettement plus courte.

Chapitre 7

Opérations abstraites des Octogones creux

Nous présentons dans ce chapitre toutes les opérations autres que la réduction qui ont été utilisées lors d'analyses statiques avec les octogones creux.

7.1 Union

L'union de deux octogones creux réduits représentés par des graphes G_1 et G_2 identiques, excepté le poids de leurs arcs, se réalise en effectuant un max entre leurs fonctions de pondération w. Sa complexité est $\mathrm{O}(E)$, donc $\mathrm{O}(N^2)$ dans le pire des cas si l'on considère toutes les contraintes comme dans le treillis des octogones.

Algorithm 7 Union $(G_1 = (V, E, w_1), G_2 = (V, E, w_2)) \rightarrow G$

if $G_1 = vide$ **then**
  Return G_2 ;
end if
if $G_2 = vide$ **then**
  Return G_1 ;
end if
for all $e \in E$ **do**
  $w_G(e) = \max(w_1(e), w_2(e))$;
end for
Return G ;

7.2 Intersection

L'intersection de deux octogones creux réduits représentés par des graphes G_1 et G_2 identiques excepté le poids de leurs arcs se réalise en effectuant un min entre leurs fonctions de pondération w. Cependant on doit réduire l'octogone creux en final car il n'est pas forcément minimal. La complexité de cette opération est la complexité de la réduction.

Algorithm 8 Intersection $(G_1 = (V, E, w_1), G_2 = (V, E, w_2)) \rightarrow G$

if $G_1 = vide$ ou $G_2 = vide$ **then**
  Return *vide* ;
end if
for all $e \in E$ **do**
  $w_G(e) = \min(w_1(e), w_2(e))$;
end for
G :=Réduire(G) ;
Return G ;

7.3 Inclusion

Un octogone creux G_1 est inclus dans un autre G_2 tous deux réduits si et seulement si pour tout arc e, $w_1(e) \leq w_2(e)$.

Algorithm 9 Inclusion $(G_1 = (V, E, w_1), G_2 = (V, E, w_2)) \rightarrow booleen$

for all $e \in E$ **do**
  if $(w_1(e) > w_2(e))$ **then**
    Return Faux ;
  end if
end for
Return Vrai ;

7.4 Test d'égalité

La représentation d'un octogone creux est canonique s'il est réduit. Deux octogones creux réduits sont égaux si et seulement si leurs fonctions w sont identiques.

Algorithm 10 Egale $(G_1 = (V, E, w_1), G_2 = (V, E, w_2)) \rightarrow$ booléen

for all $e \in E$ **do**
  if $(w_1(e) \neq w_2(e))$ **then**
    Return Faux ;
  end if
end for
Return Vrai ;

7.5 Projection

La projection π d'une variable est une opération qui enlève les contraintes sur cette variable $\pi_x(Oct_{x,y,z}) = Oct_{x',y,z}$ tel que $\exists x.Oct_{x,y,z} \wedge x' \in \mathbb{R}$. Elle est utilisée quand la valeur d'une variable est lue par exemple.

7.6 Evaluation de la borne supérieure d'une expression

Pour abstraire les affectations et les conditions, un problème récurrent est de borner une expression. Par exemple, pour l'affectation $x := -y+2z-10$, on veut borner x dans la contrainte

Algorithm 11 Projection $(G = (V, E, w), x \in Var \rightarrow (G = (V, E, w)$

$w(x^+, 0) := +\infty;$
$w(x^-, 0) := +\infty;$
for all $v \in succ(x^+)$ **do**
  $w(x^+, v) := +\infty;$
end for
for all $v \in succ(x^-)$ **do**
  $w(x^-, v) := +\infty;$
end for

$x \leq c$ en affectant à c la valeur maximale que peut prendre l'expression $-y + 2z - 10$. On utilise ici la méthode sur les intervalles pour cette évaluation.

Méthode sur les intervalles Cette méthode pour évaluer la borne supérieure d'une expression linéaire ou non est simple et peu coûteuse. Pour les expressions linéaires à coefficients dans $\{-1, 1\}$, cela donne : $\sup(\sum_{i=1}^{n} s_i x_i + a) = \sum_{i=1}^{n} w(\langle s_i, x_i \rangle, \langle 0, 0 \rangle) + a$.

7.7 Affectation

On réalise l'opération abstraite associée à une affectation en projetant la variable affectée dans l'octogone creux. On calcule ensuite les bornes de l'expression en partie droite de l'affectation pour les donner à la variable affectée. Cette opération pourra être améliorée par la suite pour prendre en compte plus précisément les affectations inversibles ($x := x + \ldots$), leur effet en analyse avant, arrière et pour récupérer des informations sur les couples de variables comme cela a été fait pour les conditions.

Algorithm 12 Affectation $x := \sum_{i=1}^{n} k_i y_i + a$

$Oct := \pi_x(Oct);$
$w(x^+, 0) := \sup(\sum_{i=1}^{n} k_i y_i + a)$
$w(x^-, 0) := \sup(-\sum_{i=1}^{n} k_i y_i - a)$
Réduire(G) ;

7.8 Intersection avec une condition

Pour traiter une condition d'un programme, nous intersectons le domaine abstrait avec la condition sous forme de contrainte. Pour cela nous calculons toutes les bornes imposées par la condition sur toutes les variables et couples de variables présentes dans la condition. Cette opération nécessite une réduction en final. Le coût le plus important est donc la complexité de l'opération de réduction.

Algorithm 13 Condition ($\sum_{i=1}^{n} a_i x_i \leq c$) avec $a_{\{1..n\}} \in \mathbb{R}$

for all i de 1 à n **do**
  $s_i := signe(a_i)$;
  $w(\langle s_i, x_i \rangle, \langle 0, 0 \rangle) := \min(w(\langle s_i, x_i \rangle, \langle 0, 0 \rangle),$
    $\sup(c - \sum_{k=1}^{n} a_k x_k - a_i x_i))/a_i$
  for all j de $i+1$ à n **do**
    $s_j := signe(a_j)$;
    if $(\langle s_i, x_i \rangle, \langle -s_j, x_j \rangle) \in E$ **then**
      $c_1 := max(|a_i|, |a_j|)$;
      $c_2 := min(|a_i|, |a_j|)$;
      $w(\langle s_i, x_i \rangle, \langle -s_j, x_j \rangle) := \min(w(\langle s_i, x_i \rangle, \langle -s_j, x_j \rangle),$
        $\sup(c - \sum_{k=1}^{n} a_i x_i + s_i c_1 x_i + s_j c_1 x_j),$
        $\sup(c - \sum_{k=1}^{n} a_i x_i + s_i c_2 x_i + s_j c_2 x_j))$;
    end if
  end for
end for
Réduire(G) ;

Conclusion

Le nouveau domaine des octogones creux a fait l'objet d'une implémentation. Le langage choisi est C, ce qui permettra d'interfacer facilement la librairie avec différents langages (Java, OCaml, ...).

Le graphe est implémenté sous la forme de listes d'adjacence, une structure de donnée classique et efficace pour la représentation des graphes. Nous avons choisi un tas binaire pour implémenter la file de priorité dans l'algorithme de Dijkstra. La complexité asymptotique est moins bonne que pour les tas binaires, mais aussi plus simple. Ce choix ne multiplie que par $\log(N)$ la complexité du domaine (Voir comparatif chapitre 6).

Toutes les opérations décrites dans ce document ont été implémentées : réduction, borne supérieure, borne inférieure (intersection), intersection avec une contrainte linéaire ou non linéaire, affectation d'une variable par une expression linéaire ou non. Ceci permet donc d'utiliser le domaine pour le modèle de programme que l'on s'est fixé au chapitre 1. Les algorithmes de graphes nécessaires ont été réimplémentés pour coller à nos structures de données, optimisées pour une faible occupation mémoire. La taille de cette implémentation est d'environ 1300 lignes.

Il existe plusieurs pistes pour améliorer et étendre ces recherches. Tout d'abord, l'algorithme de Dijkstra, utilisé dans la réduction des octogones creux, peut être optimisé pour de nombreux cas (réduction des contraintes ($v \leq c$), graphe à composantes fortement connexes (chapitre 6), ...). Ensuite, dans le cas de variables de type entier, l'opération de réduction dans les octogones est en $O(N^4)$. L'étude des entiers dans le domaine des octogones creux est à étudier. Enfin, l'application d'un produit réduit sur les octogones s'est révélé réalisable et permet une réduction de la complexité de l'analyse. Il est sans doute intéressant d'appliquer ce produit réduit dans d'autres domaines notamment les polyèdres, où la vision du produit réduit comme un polyèdre creux n'est pas possible.

Nous n'avons malheureusement pas eu le temps d'expérimenter l'utilisation de ce domaine en analyse de programmes, ni d'étudier des heuristiques pour déterminer la répartition des variables du programme en différents paquets.

Il est prévu, à terme, d'utiliser cette bibliothèque dans l'outil Nbac de Bertrand Jeannet, qui sert à valider des programmes synchrones et des systèmes hybrides. Cette intégration pourra permettre également d'expérimenter ce domaine pour mieux saisir ses capacités et ses limites. Par ailleurs, ce domaine pourra trouver d'autres utilisateurs, notamment au sein de l'ACI Sécurité Informatique APRON (Analyse de PROgrammes Numériques) à laquelle participe Bertrand Jeannet.

Bibliographie

[BCC+03] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jerome Feret, and David Monniaux Xavier Rival Laurent Mauborgne, Antoine Miné. A static analyser for large safety-critical software. In *Programming language design and implementation (PLDI)*, San Diego, California, USA, 2003.

[CC76] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *2nd Int. Symp. on Programming*. Dunod, Paris, 1976.

[CC77] P. Cousot and R. Cousot. Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM Symposium on Principles of Programming Languages, POPL'77*, Los Angeles, January 1977.

[CC92a] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(2–3) :103–179, 1992. (The editor of Journal of Logic Programming has mistakenly published the unreadable galley proof. For a correct version of this paper, see `http://www.di.ens.fr/~cousot`).

[CC92b] P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In M. Bruynooghe and M. Wirsing, editors, *PLILP'92*, volume 631 of *LNCS*, Leuven (Belgium), January 1992.

[CH78] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *5th ACM Symposium on Principles of Programming Languages, POPL'78*, Tucson (Arizona), January 1978.

[CLR94] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to algorithms*. MIT Press, 1994.

[HPR97] N. Halbwachs, Y.E. Proy, and P. Roumanoff. Verification of real-time systems using linear relation analysis. *Formal Methods in System Design*, 11(2), August 1997.

[Min01] A. Miné. The octagon abstract domain. In *AST'01 in Working Conference on Reverse Engineering 2001*. IEEE CS Press, October 2001.

[Min04] Antoine Miné. *Domaines numériques abstraits faiblement relationnels*. PhD thesis, École Normale Supérieure (Paris), December 2004.

[SKH02] A. Simon, A. King, and J.M. Howe. Two variables per linear inequality as an abstract domain. In *Int. Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'02)*, volume 2664 of *LNCS*, 2002.

[SSM05] Henri B. Sipma Sriram Sankaranarayanan and Zohar Manna. Scalable analysis of linear systems using mathematical programming. In *Verification, Model-Checking, and Abstract Interpretation (VMCAI 2005)*, Gières (France), 2005.

[Yov98] Sergio Yovine. Model checking timed automata. In *Lecture Notes in Computer Science 1494*, Gières (France), 1998.