Implémentation des domaines entiers dans le solveur de contraintes AbSolute

Mathieu Vavrille
ENS de Lyon, Stage de L3
sous la direction de Charlotte Truchet à l'université de Nantes

27 Août 2017

1 Introduction

La programmation par contraintes est un paradigme de programmation permettant de résoudre de grandes instances de problèmes combinatoires, comme n-dames, les carrés magiques, les sudokus, SAT, ..., en séparant la modélisation du problème de sa résolution.
De nombreux solveurs de contraintes existent (Chip V5, Choco, gecode, Ilog Solver par exemple) mais ils sont tous spécialisés dans un type de variable : réelles ou entières. Chaque type de variables est différent : les variables réelles peuvent faire intervenir des fonctions d'analyse (comme sinus, exponentielle, ...) et les variables entières ont des contraintes particulières (contraintes globales). Chacun des domaines a des algorithmes de résolution spécifiques et les solveurs actuels sont spécialisés dans un seul type de domaine.

Le solveur AbSolute sur lequel j'ai travaillé [Pelleau et al., 2013] a pour but d'être le premier solveur mixte dans sa conception. Pour cela il utilise le formalisme de l'interprétation abstraite [Cousot and Cousot, 1977] pour représenter les domaines des variables. Dans AbSolute, les domaines (ensembles de valeurs possibles que peuvent prendre les variables du problème) sont abstraits au sens de l'interprétation abstraite, ce qui permet d'une part d'avoir des domaines plus expressifs qu'habituellement en contraintes, et d'autre part de résoudre simultanément dans plusieurs domaines abstraits. Dans l'état courant d'AbSolute avant le début de ce stage, les variables réelles étaient les plus développées et plusieurs domaines étaient déjà implémentés : produits cartésiens, octogones et polyèdres (un exemple de solution avec deux domaines différents est donné en figure 1).

Le but du stage a été d'implémenter un solveur de contraintes à variables discrètes dans l'optique de l'intégrer dans le solveur actuel. Ceci permettra d'avoir une gestion spécifique des contraintes réelles et entières, au moyen des outils fournis par l'interprétation abstraite comme le produit réduit qui permet de combiner des domaines.
Ce solveur mixte sera le premier de son genre. Les solveurs actuels peuvent résoudre des problèmes mixtes mais sous une forme détournée : assimiler les entiers à des réels ou réciproquement (perdant ainsi les algorithmes de résolution spécifiques à tel ou tel type), ou organiser des coopérations de solveurs qui sont difficiles à mettre en place, ne serait-ce que pour des raisons techniques.

Table des matières

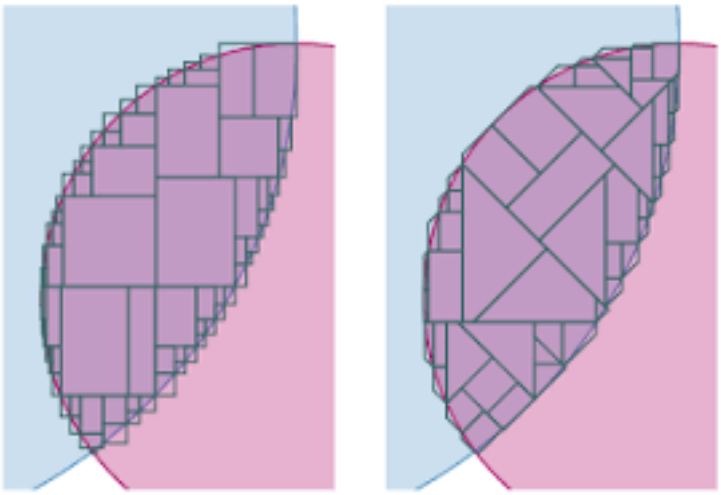

FIGURE 1 – Solution d'un problème à variables réelles avec des boites ou des octogones

2 Programmation par contraintes

Dans cette section, j'introduis les notions de programmation par contraintes qui sont utiles à la compréhension de mon travail de stage. Pour une présentation plus détaillée, on peut se reporter à [Rossi et al., 2006], dont cette présentation s'inspire.

2.1 Problème de satisfaction de contraintes

Définition 1 (Contrainte)**.** Soit $r \in \mathbb{N}$, Une contrainte est une relation définie sur des variables $X(c) = (x_{i_1}, \ldots, x_{i_r})$. L'ensemble des solutions de c est un sous-ensemble de $\mathbb{Z}^r$ qui contient les tuples satisfaisant c. r est appellé l'arité de la contrainte.

Une contrainte peut être spécifiée en extension (par la liste de ses tuples) ou en intension (par une formule qui la caractérise). Les contraintes d'arité 2 sont dites binaires. Une contrainte globale est une contrainte définie par une formule d'arité arbitraire.

Exemple.
- $x + y < 3$
- $x * y + z \neq t$
- *La contrainte globale* $all\_different(x_1, x_2, x_3) \equiv (v_1 \neq v_2 \wedge v_1 \neq v_3 \wedge v_2 \neq v_3)$ *autorise les tuples dont toutes les valeurs sont différentes. Cette contrainte peut être étendue à un nombre arbitraire de variables.*

Définition 2 (CSP)**.** Un problème de satisfaction de contraintes (CSP) est un triplet (X, D, C) où :

— $X = (x_1, \ldots, x_n)$ un ensemble de variables
— $D = (D_1, \ldots, D_n)$ des ensembles de valeurs pour chaque variable ($x_i \in D_i$)
— $C = (C_1, \ldots, C_m)$ un ensemble de contraintes

Par la suite on suppose que les contraintes ont au moins 2 variables (les contraintes unaires reviennent à diminuer l'ensemble de définition de la variable impliquée). Un problème de satisfaction de contraintes peut se représenter comme un hypergraphe où chaque variable est un sommet et chaque contrainte est une hyperarête.

Pour simplifier les notations, on utilisera les lettres x pour les variables de X (le nom qu'on leur a donné dans les contraintes) et v pour les valeurs qu'elles peuvent prendre (les éléments dans $D(x)$ pour $x \in X$). Par exemple on dira qu'on affecte la valeur v_1 à la variable x_1.

Définition 3 (Instanciation)**.** Une instanciation I sur un ensemble de variables $Y = (x_1, \ldots, x_k)$ est une affectation des valeurs $v_1, \ldots, v_k$ aux variables. On dit qu'une affectation est :

— valide si pour tout $x_i \in Y, I[x_i] \in D(x_i)$
— localement consistante si pour toute contrainte $c \in C$ telle que $X(c) \in Y$, $I[X(c)]$ satisfait c.
— une solution du problème si $Y = X$ et I est localement consistante et valide.

Résoudre un problème de satisfaction de contraintes peut prendre plusieurs formes :

— Trouver toutes les solutions
— Trouver une solution
— Montrer qu'il n'y a pas de solutions
— Trouver la solution qui maximise/minimise une certaine fonction

Dans la suite, on cherchera en général à trouver toutes les solutions à un problème (les méthodes employées dans les autres cas sont similaires et emploient des algorithmes très proches). La difficulté vient du fait que l'espace des solutions a une taille exponentielle (avec n variables telles que $|D(x)| = m$, l'espace a une taille m^n). Il faut donc trouver des méthodes pour réduire cet espace, et des heuristiques pour simplifier le problème et accélérer la résolution.

2.2 Consistances et propagation

La recherche des solutions en parcourant exhaustivement toutes les instanciations possibles étant trop coûteuse, on essaie de trouver des valeurs que certaines variables ne peuvent pas prendre (au vu des contraintes). Il y a différentes méthodes pour trouver de telles valeurs, certaines vont trouver beaucoup de valeurs à supprimer mais seront coûteuses en temps, ou inversement. Il s'agit de trouver un compromis entre le temps pris pour réduire l'espace de recherche et le nombre de valeurs que l'on va supprimer.

2.2.1 Consistance d'arc

La forme de consistance la plus connue est la consistance d'arc (AC). Intuitivement, il s'agit de savoir si pour une contrainte donnée, une valeur apparaît dans un tuple de cette contrainte.

Définition 4 (Consistance d'arc)**.**

— Pour $c \in C$ une contrainte, $x_i \in X$ une variable, $v_i \in D(x_i)$ est consistante avec c sur D ssi il existe un tuple τ satisfaisant c et tel que $v_i = \tau[x_i]$. Un tel tuple est appelé support pour (x_i, v_i) sur c.
— Un domaine D est arc-consistant sur c pour x_i ssi toutes les valeurs dans $D(x_i)$ sont consistantes avec c sur D.
— Un CSP est arc-consistant ssi D est arc-consistant pour toute variable sur toute contrainte.

La consistance d'arc est en général NP-difficile à assurer pour des contraintes r-aires (d'arité r). L'algorithme a une complexité en $O(d^r)$ avec r l'arité de la contrainte et d la taille maximale des domaines des variables. Cependant, dans beaucoup de problèmes, l'arité des contraintes n'est pas trop élevée : dans ce cas imposer la consistance d'arc sur toutes les contraintes peut se révéler très efficace et supprimer beaucoup de valeurs inconsistantes.

Algorithm 1 AC3

```
function REVISE(x_i :variable, c :contrainte)                    ▷ Consistance
    CHANGE ← false
    for v_i ∈ D(x_i) do
        if v_i n'a pas de support pour c then
            Supprimer v_i de D(x_i)
            CHANGE ← true
        end if
    end for
    return CHANGE
end function

function AC3((X, D, C) : CSP)                                    ▷ Propagation
    Q ← {(x_i, c)|c ∈ C, x_i ∈ X(c)}
    while Q ≠ ∅ do
        (x_i, c) ← pop(Q)
        if Revise(x_i, c) then
            if D(x_i) = ∅ then return false
            else
                Q ← Q ∪ {(x_j, c')|c' ∈ C, c' ≠ c, x_i, x_j ∈ X(c), i ≠ j}
            end if
        end if
    end while
    return true
end function
```

2.2.2 Propagation de la consistance

Les algorithmes de consistance ont habituellement deux parties. La première est la recherche de valeurs inconsistantes dans une contrainte, et la deuxième est la propagation de la suppression des valeurs. En effet, la suppression d'une valeur peut rendre d'autres valeurs inconsistantes pour d'autres contraintes.

AC3 :

L'algorithme le plus connu pour assurer la consistance d'arc d'un CSP a été proposé par Mackworth [Mackworth, 1977] sous le nom AC3 [1]. L'idée est de garder une liste de couples (x_i, c) pour toutes les variables pour lesquelles on n'est pas sûr que $D(x_i)$ soit consistant avec c. Pour chaque variable d'un couple on parcourt son domaine à la recherche de valeurs inconsistantes, qu'on supprime. Si on a supprimé une valeur, de nouvelles variables sont peut-être devenues inconsistantes, on augmente donc la liste des couples variables-valeurs concernés.

Toute la complexité (en temps) de l'algorithme réside dans la recherche du support, qui se fait par un parcours de tous les tuples possibles. On va seulement appliquer `Revise` tant qu'une valeur a été supprimée, jusqu'à obtenir un point fixe (qu'aucune variable ne change).

Améliorations :

Il existe de nombreuses améliorations de cet algorithme, toutes viennent ajouter un principe de propagation plus intelligent pour ne pas avoir à faire de calculs inutiles (vérifier une contrainte trop souvent, rechercher plusieurs fois des supports pour la même variable, ...).

On se place dans le cadre de contraintes binaires, dont les ensembles de variables sont différents ($X(c) \neq X(c'), \forall c, c' \in C$). L'algorithme peut se généraliser à des contraintes r-aires, mais pour une facilité d'écriture, seules les contraintes binaires sont présentées ici.

L'algorithme AC4 [Mohr and Henderson, 1986] va stocker des informations pour chaque contrainte pour ne pas avoir à recalculer de supports à chaque fois. On garde en mémoire le nombre de supports d'une valeur pour chaque contrainte dans un tableau `counter`$[x_i, v_i, x_j]$ (où $\{x_i, x_j\} = X(c), c \in C$), et dans une liste $S[x_i, v_i]$ les valeurs supportées par (x_i, v_i). Ainsi, lors de la suppression d'une valeur, on connaît toutes les valeurs (grâce à S) qui n'ont potentiellement plus de support. On décrémente leur compteur et s'il devient nul, la valeur est devenue inconsistante avec une contrainte et on la supprime à son tour.

L'algorithme AC6 [Bessiere, 1994] continue à améliorer le même algorithme en se souvenant

d'*un seul*. L'algorithme va stocker moins de données, ce qui donne une complexité en espace plus faible que AC4.

L'algorithme AC2001 [Bessière et al., 2005] repose sur le même principe mais se souvenant uniquement du *plus petit* support dans la liste S (pour l'ordre lexicographique). Ainsi, la recherche de nouveaux supports se fera de manière plus rapide, car on pourra continuer dans l'ordre lexicographique au lieu de recommencer dès le début à chaque fois.

2.2.3 Contraintes globales

La définition des contraintes étant très large, on peut créer de nombreuses contraintes différentes. Il vient tout de suite à l'esprit les contraintes linéaires (équations, inéquations), non linéaires. Ici il est question des contraintes globales :

Définition 5 (Contrainte globale)**.** Une contrainte globale est une contrainte qui est définie sur un nombre arbitraire de variables.

Ces contraintes permettent de représenter des problèmes très particuliers qui sont difficilement représentables par une autre expression.

La contrainte globale la plus connue est la contrainte `all_different`$(x_1, \ldots, x_n)$ qui impose que dans une instanciation des variables, toutes les valeurs de $x_1, \ldots, x_n$ soient différentes. Cette contrainte peut être ré-écrite avec des contraintes de différence : $\forall i < j, x_i \neq x_j$, mais la résolution est plus rapide en l'écrivant sous la forme compacte.

Il y a une multitude de contraintes globales, un lecteur curieux pourra regarder le catalogue des contraintes globales [Beldiceanu et al., 2012] qui en décrit des centaines.

Les contraintes globales ont le désavantage de faire intervenir un nombre élevé de variables (habituellement les contraintes usuelles ne font pas intervenir un trop grand nombre de variables). Une recherche normale des supports peut être très coûteuse, on cherche donc des algorithmes plus efficaces pour pouvoir assurer la consistance de ces contraintes. Souvent pour les contraintes globales un tel algorithme existe.

All_different : Il existe un algorithme efficace pour assurer la consistance de la contrainte `all_different` proposé par Régin [Régin, 1994]. Celui-ci fait intervenir des notions de théorie des graphes dont les définitions sont rappellées :

Définition 6 (Couplage)**.** Soit $G = (V, E)$ un graphe, un couplage M est un ensemble d'arêtes ($M \subseteq E$) tel que chaque sommet du graphe apparaît dans au plus une arête du couplage.
Un sommet lié (resp. non lié) à une arête du couplage M est dit M-saturé (resp. M-insaturé).
Un couplage est dit couvrant $X \subseteq V$ si tous les sommets de X sont M-saturés.
Un couplage est dit maximum si il a le plus grand cardinal possible.
Un chemin dans un graphe est dit M-alternant si il prend tour à tour une arête du couplage et une arête qui n'est pas dans le couplage (il alterne).

Définition 7 (Graphe biparti)**.** Un graphe $G = (V, E)$ est dit biparti si $V = X \cup Y$ et $E \subseteq X \times Y \cup Y \times X$.

Pour filtrer (assurer la consistance) la contrainte, on utilise une représentation de graphe :

Définition 8 (Graphe de valeurs)**.** Soit X un ensemble de variables.
On définit le graphe $G = (X \cup D, E)$ où $X = X(c)$ et $D = \bigcup_{x_i \in X} D(x_i)$ toutes les valeurs pouvant être prises par les variables et $E = \{(x_i, v_j) | v_j \in D(x_i)\}$. C'est un graphe biparti qui lie chaque variable aux valeurs qu'elle peut prendre.

En utilisant ce graphe on a une condition nécessaire et suffisante sur les instanciations satisfaisant la contrainte `all_different` :

Théorème 1. *Soit $X = \{x_1, \ldots, x_n\}$, et G le graphe de valeurs de X.*
Alors $(d_1, \ldots, d_n) \in all\_different(x_1, \ldots, x_n)$ si et seulement si $M = \{(x_1, d_1), \ldots, (x_n, d_n)\}$ est un couplage dans G.

Par exemple, pour le problème associé à la figure 2 ($x \in \{1, 2, 3\}, y, z \in \{2, 3\}$), l'arête $(x, 3)$ ne fait pas partie d'un couplage, et donc est inconsistante, $x = 3$ ne peut pas apparaître dans une solution.

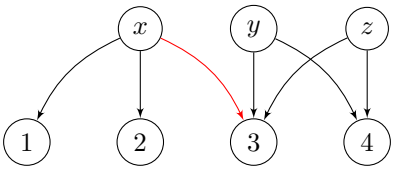

FIGURE 2 – Exemple de graphe de valeurs

Démonstration. Par définition du couplage, du graphe de valeurs et de la contrainte. □

Pour assurer la consistance de la contrainte il suffit maintenant de trouver toutes les valeurs qui apparaissent dans un couplage couvrant $X = X(c)$ (ce qui revient à un couplage maximum) avec $c = all\_different(x_1, \ldots, x_n)$. Pour cela on a encore un théorème qui nous donne une condition nécessaire et suffisante.

Théorème 2. *Soit G un graphe et M un couplage maximum dans G, une arête e appartient à un couplage maximum ssi $e \in M$ ou e est sur un chemin M-alternant de taille paire partant d'un somment M-insaturé ou e est dans un cycle M-alternant de taille paire.*

En utilisant ce théorème on peut donner un algorithme pour trouver toutes les arêtes d'un graphe biparti apparaissant dans un couplage maximum (et ainsi assurer la consistance de la contrainte).

1. On cherche un couplage maximum dans $G = (X \cup D, E)$ le graphe de valeurs
2. On construit le graphe $G' = (X \cup D, E')$ où $E' = \{(x, d) | x \in X \wedge (x, d) \in M\} \cup \{(d, x) | x \in X \wedge (x, d) \in E \backslash M\}$
3. Toutes les arêtes de G' apparaissant dans une même composante fortement connexe appartiennent à un cycle M-alternant
4. Toutes les arêtes trouvées par parcours du graphe partant d'un somment M-insaturé appartiennent à un chemin M-alternant

Ainsi il suffit de supprimer toutes les arêtes qui ne sont pas dans un couplage et on a assuré la consistance d'arc de la contrainte (en effet, si une arête appartient à un couplage alors ce couplage donne un support de la contrainte).

Maintenant on cherche à connaître la complexité de cet algorithme, on note $m = |E|$ et $n = |X|$ pour (X, E) le graphe de valeurs :

1. Rechechre du couplage maximum : il existe plusieurs algorithme pour le faire, certains sont très efficaces. Avec l'algorithme de Hopcroft-Karp (des chemins améliorants) on peut le trouver en $O(\sqrt{m}n)$
2. La création du graphe se fait en $O(m + n)$
3. La recherche des composantes fortement connexes se fait en $O(m + n)$ avec l'algorithme de Kosaraju ou Tarjan
4. Les parcours se font en temps $O(m + n)$ au total car on ne parcours pas 2 fois les mêmes sommets.

L'algorithme se fait donc en $O(\sqrt{m}n)$. On peut aussi regarder ce qui se passe lors de la suppression d'une valeur (avec la consistance d'une autre contrainte). Dans ce cas on essaie de ne pas tout recalculer et de rechercher à nouveau les couplages de manière incrémentale.
Si on a supprimé une arête du couplage de départ, retrouver un nouveau couplage se fait en $O(n + m)$ (un seul parcours de graphe est nécessaire). Ensuite on refait les étapes 3 et 4. On a donc un algorithme incrémental pour la consistance de la contrainte en $O(n + m)$

2.3 Backtrack

Toutes ces étapes de consistance permettent de réduire le domaine de recherche, cependant elles ne donnent pas les solutions.

Pour cela il faut faire la recherche à proprement parler, et la méthode la plus simple est un backtrack [2].

Algorithm 2 Backtrack

```
function BACKTRACK(P = (X, D, C) : CSP)
    P ← PROPAGATION(P)
    if CONDITION_ARRET(P) ∧ IS_SOLUTION(P) then
        AFFICHER(P)
    else
        for csp in SPLIT(P) do
            BACKTRACK(csp)
        end for
    end if
end function
```

Le principe est très simple : on commence par assurer la consistance d'arc du problème pour le simplifier. Ensuite si on a une solution, on peut l'afficher ou la stocker, mais si on n'a pas trouvé une solution, il faut continuer la recherche. Pour cela on divise le problème en sous-problèmes (`split`), et on applique récursivement le backtrack. Dans le cas des variables entières, la méthode générale pour cette fonction `split` est de choisir une variable, et de lui affecter une valeur de son domaine (et faire la recherche pour toute affectation de valeur possible).

Ici interviennent plusieurs heuristiques pour choisir les variables : la première idée est de choisir la variable qui a le plus petit domaine (méthode `dom`), pour avoir un arbre de recherche plus petit (faire moins d'appels récursifs).
On peut aussi choisir la variable qui intervient dans le plus de contraintes (méthode `deg`), pour ensuite avoir une consistance qui va supprimer plus de valeurs.
Enfin, on peut faire le quotient de la taille du domaine par le degré, et prendre la variable qui a la plus petite valeur (méthode `dom/deg`), ou même associer un poids à chaque contrainte (certaines vont plus filtrer des valeurs) et faire un quotient pondéré (méthode `dom/wdeg`).

2.4 Améliorations et heuristiques

Dans cette partie on va voir quelques améliorations possibles, et des heuristiques différentes. Les notions sont définies rigoureusement dans "The handbook of constraint programming" [[Rossi et al., 2006]] et ici il sera uniquement fait un survol pour avoir une idée des autres choix possibles.

2.4.1 Autres consistances

Dans les cas où la consistance d'arc est trop coûteuse à imposer, il peut être utile d'avoir une définition plus lâche de consistance mais qui sera plus rapide à assurer. C'est le cas de la consistance de bornes (BC).

Définition 9 (Consistance de bornes)**.** On dit qu'une contrainte $c \in C$ est consistante aux bornes si pour toute variable $x_i \in X(c)$, $(x_i, \min(x_i))$ et $(x_i, \max(x_i))$ appartiennent à un support de c.

La consistance de bornes est plus rapide à assurer que la consistance d'arc, malheureusement dans le cas général elle va supprimer moins de valeurs.

On peut aussi penser à des formes de consistances plus restrictives, qui en contrepartie seront plus coûteuses. Il existe beaucoup de consistances, en voici une liste restreinte :

- Consistance de chemins
- k-consistance, i-j-consistance
- Consistance de chemins restreinte
- Consistance de chemins inverse

Certaines de ces consistances vont permettre de supprimer des valeurs (comme la consistance d'arcs, ou de bornes), d'autres vont supprimer des couples (n-uplets) de valeurs : on pourra assurer que 2 valeurs ne peuvent pas intervenir *en même temps* dans une solution, sans pour autant qu'elles soient chacune inconsistante.

2.4.2 Autres recherches de solutions

L'algorithme de backtrack présenté un peu plus haut est la version la plus simple possible. Ici aussi il y en a d'autres qui selon les applications peuvent être meilleurs.

Dans celui qui a été présenté, on copie toute la structure de donnée à chaque nœud de l'arbre (du parcours des valeurs). Selon les méthodes de résolutions utilisée, il est possible de ne pas copier la structure de données : pour cela il faut pouvoir retrouver l'état précédent du programme. Ainsi on économise de l'espace mémoire mais en contrepartie, le calcul de l'ancien état peut être coûteux. On peut faire un compromis entre les deux méthodes en copiant l'état du programme seulement à certaines étapes, et en recalculant l'état dans les autres cas.

Toutes les heuristiques présentées pour le choix des variables ont un seul désavantage : le programme choisit une heuristique et l'applique tout le temps de la même manière. Cela peut-être un problème quand il s'avère que le problème est particulier et arrive tout le temps dans le pire cas des heuristiques (ce qui les rend inutiles). Pour corriger cela, on peut définir des heuristiques dynamiques, qui s'adaptent au cours de la résolution, ou bien randomiser notre algorithme, et de plusieurs manières différentes : on peut choisir aléatoirement la variabre à instancier [Harvey, 1995], on peut choisir aléatoirement l'heuristique à appliquer, on peut randomiser l'algorithme à partir d'un certain moment. Cette randomisation permet de se rapprocher du cas moyen de l'algorithme, et ne pas tomber dans le pire cas à chaque fois, ce qui pourrait être très coûteux.

Il a déjà été question de simplifier la consistance supprimant moins de valeurs, et ainsi accélérer le processus. Dans le backtrack se pose une nouvelle question : faut-il assurer la consistance à chaque étape ? Pour accélérer, on peut choisir de ne pas assurer la consistance de toutes les contraintes. Avec la méthode du "Forward Checking", on va seulement assurer la consistance des contraintes qui font intervenir la variable qui vient d'être instanciée (par le backtrack).

3 Travail de stage

Ce qui a été présenté sont les méthodes de résolution des problèmes de satisfaction de contraintes entiers (discrets). Le solveur Absolute à pour l'instant été fait uniquement pour les réels, car la priorité à été mise sur le fait d'implémenter de nouveaux domaines abstraits (boites, octogones, polyèdres). Le langage de programmation utilisé est Ocaml.

3.1 Choix d'orientation du stage

La partie implémentation du stage a été divisée en deux parties : premièrement j'ai utilisé l'implémentation d'Absolute pour coder les intervalles entiers, ensuite je me suis écarté d'Absolute pour implémenter les algorithmes spécifiques aux entiers.

Absolute :

Me plonger dans le code d'AbSolute m'a permis de découvrir quelques améliorations possibles pour plus de généralité. En effet, les domaines abstraits existant dans AbSolute étaient en pratique dédiés aux réels. En collaboration avec Charlotte Truchet et Ghiles Ziat, j'ai proposé une une version plus simple, et surtout plus générique, des domaines dans le solveur. Les domaines abstraits définis maintenant sont adaptés aussi bien aux réels, qu'aux entiers ou potentiellement à d'autres types de variables.

Les problématiques qui se posent pour le calcul sur les nombres réels (représentés en flottants) sont essentiellement celles des arrondis. Dans le solveur actuel, les calculs sont garantis en faisant des sur-approximations pour ne jamais perdre de solutions à cause de la représentation inexacte des flottants en machine.

Généralisation :

Dans un second temps j'ai implémenté le solveur discret à partir de zéro (en réutilisant le lexer/parser d'Absolute). Parmi tous les algorithmes et heuristiques qui ont été présentés précédemment, il a fallu faire des choix. Ces choix ont été dictés par plusieurs raisons :

— contraintes d'arité supérieures à 2 : les algorithmes sont plus simples à écrire pour des contraintes binaires, mais dans le cas général c'est plus difficile.

— cohérence avec le reste du solveur : le but est de construire un solveur mixte. Il faut donc avoir une interface simple et facilement modifiable/extensible pour de futures améliorations.
— avoir une complexité raisonnable : les algorithmes présentés ont des complexités différentes et selon leur utilisation et la structure du code certains seront meilleurs que d'autres.

On veut aussi pouvoir gérer précisément la propagation (commencer par imposer la consistance sur les contraintes "faciles").

L'implémentation d'Absolute ne permettait pas de faire tout cela, c'est pourquoi il a fallu implémenter un solveur de contraintes discret à partir de zéro. J'ai choisi mes propres structures de données et les algorithmes qui seront executés (et que j'ai du implémenter).

3.2 Dans Absolute

Dans un premier temps dans le stage, j'ai utilisé l'implémentation d'Absolute pour faire une version très simple d'un solveur spécifique discret.

3.2.1 Structure

Absolute est codé avec le langage Ocaml, et utilise l'interface fonctorisée du langage (modules paramétrés). Il se sépare en plusieurs parties :

— parser/lexer
— affichage de la solution : en format tikz, en graphique, et 3D quand il y a 3 variables
— librairies : tous les modules utiles à la résolution
— solveur qui utilise tous les autres modules

Le parser/lexer prend un programme écrit sous la forme :

```
init {
  int x = [5;12];
  real y = [-10;50];
}

constraint {
  x+y<5;
  max(x, y) > 10;
}
```

Le problème est ensuite rendu utilisable par la machine.

L'affichage permet plusieurs paramètres : on peut générer une sortie en format tikz (pour une utilisation par un document latex), un affichage graphique ou encore un format .obj pour les visualisations 3D.

Maintenant il faut expliquer la structure du code. L'algorithme de résolution est l'algorithme HC4 (que je ne vais pas expliquer dans les détails). C'est un algorithme de calcul sur les intervalles : on va parcourir l'arbre qui décrit la contrainte (arbre syntaxique) vers le haut puis redescendre pour réduire les intervalles des variables quand ils sont inconsistants. Il faut dont savoir faire les calculs sur les intervalles. Les modules des domaines abstraits (boites, octogones, polyèdres) prennent en paramètre un module d'intervalles pour faire les calculs.

Les intervalles sont constitués de 2 bornes. Lors de la définition d'un module d'intervalles il faut donner en paramètre un module de bornes. Les bornes peuvent être de nombres flottants et à ce moment il faut définir les opérations d'addition, multiplication avec arrondi supérieur et inférieur pour avoir des calculs exacts. Lors d'une opération sur les intervalles on va faire un arrondi à chaque fois : on va sur-approximer la solution pour ne rien perdre.

Dans mon cas il a fallu définir les bornes entières. Les opérations sur les entiers étant exactes, il n'y a pas tous les problèmes d'arrondis.

Une autre modification a été le `split` pour le backtrack. Pour les nombres réels, on divise l'intervalle en 2 par le milieux. Pour les entiers on peut énumérer les valeurs possibles. Cela a demandé une modification du code général pour pouvoir faire ce `split`.

L'implémentation des bornes entières a permis d'avoir un programme qui marche mais il restait encore un petit détail qui pouvait poser problème.

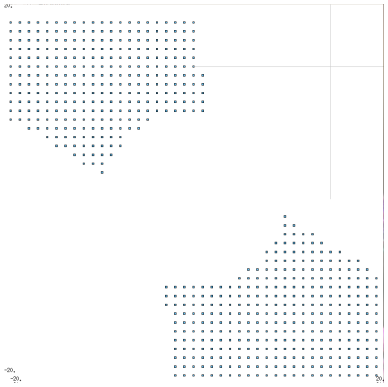

FIGURE 3 – Solution d'un problème

3.2.2 Intervalles rationnels

Un problème qui nous a fait réfléchir est le cas de la division : comment traiter les contraintes de la forme $x/y < 30$ (en oubliant le cas de la division par 0 qui est seulement un cas particulier) ? Dans les réels c'est facile, la division est bien définie mais dans les entiers ce n'est pas si simple. Nous avons fait le choix de voir la contrainte d'un point de vue abstrait, sans les problèmes de typage : c'est à dire qu'une formule est vraie ou fausse en mathématiques, une division va nous forcer à utiliser des nombres flottants.

En fait en se limitant aux divisions, on peut toujours avoir des opérations exactes en utilisant les nombres rationnels. Ce qui a été fait a alors été de créer le module des intervalles entiers mais avec des calculs intermédiaires rationnels. Lors des calculs on va utiliser les nombres rationnels (avec un module spécifique) puis on va à la fin se ramener aux entiers.
Par exemple pour la variable $x \in [0; 12]$ et la contrainte $x/3 <= 3$, les calculs vont être :

— On calcule $[0; 12]/3 = [0; 4]$
— la contrainte va nous ramener à l'intervalle $[0; 3]$
— on finit en multipliant par 3 : $x \in [0; 9]$

Cela permet de gérer les divisions, mais on voit très bien que la gestion est plus difficile que pour les réels.

Tout cela permet d'avoir des résultats graphiquement. Par exemple la figure 3.2.2 provient de la résolution du système :

— $x \in [-20; 20]$
— $y \in [-20; 20]$
— $y * x + x * x/2 + y \leq 30$
— $y * y + x * x >= 100$

Dans ce qui vient d'être présenté grâce à l'interface fonctorisée (modulaire) on n'a que peu de possibilités pour changer les algorithmes de propagation et de consistance. Les structures de données annexes nécessaires pour les contraintes globales (un graphe pour `all_different`) sont impossibles à stocker. C'est pour cela qu'il a fallu que je réfléchisse en profondeur à l'implémentation dans le cadre général.

3.3 Dans le cadre général

Maintenant on essaie d'avoir un code qui va utiliser les algorithmes spécifiques aux entiers. Les domaines des variables peuvent maintenant être n'importe lesquels : on utilise des ensembles car la pour assurer la consistance d'arc on veut pouvoir supprimer une valeur n'importe où dans le domaine. Grâce à l'interface modulaire on peut très facilement changer l'implémentation de ces ensembles. L'implémentation actuelle utilise les ensembles Ocaml, qui sont en pratique des arbres binaires de recherche, donc les opérations de recherche, suppression, se font en $O(log(n))$ avec n la taille de l'ensemble.

Il faut ensuite améliorer la représentation des problèmes de satisfaction de contraintes.

3.3.1 Problème amélioré

Pour rendre le problème plus facilement utilisable, j'ai fait quelques modifications, et ajouté des structures de données annexes.

La première chose a été d'utiliser des entiers comme identificateurs des variables du problème. Jusqu'à présent, les variables étaient utilisées sous forme de chaînes de caractères, et les opérations de recherche se faisaient donc en $O(log(n))$ avec n le nombre de variables. Maintenant on utilise un tableau, et chaque variable devient un identificateur pour accéder à la case du tableau. On modifie les contraintes pour ne plus avoir de chaînes de caractères mais des entiers pour représenter les variables.

Les contraintes sont ensuite triées selon leur "type", car le but est d'appliquer une consistance spécifique plus rapide selon les cas. Voici le type Ocaml des contraintes :

```
type qualification = Eq_lin of eq_lin_data array
        | Ineq_lin of ineq_lin_data arra
        | All_dif of matching_graph
        | Other of compteur * support
```

C'est un type somme, c'est à dire que chaque contrainte utilisera un seul des 4 constructeurs (Eq_lin, Ineq_lin, All_dif, Other) avec la structure de donnée associée. L'avantage du type somme est de pouvoir facilement ajouter des cas particuliers. C'est une des choses qu'il a fallu préserver dans la suite du code pour de futures améliorations. Pour l'instant il n'y a que ces 4 cas :

- Eq_lin correspond aux équations linéaires
- Ineq_lin aux inéquations linéaires
- All_dif est le type spécial pour la gestion de la contrainte `all_different`
- Other correspond à toutes les autres contraintes dont on n'a pas (encore) d'algorithme spécifique.

Le but est de séparer la consistance de la propagation. L'algorithme de propagation va seulement propager les suppression de valeurs aux variables devenues potentiellement inconsistantes. Passons aux algorithmes spécifiques :

Inéquations linéaires :

L'idée est tirée de l'article [Codognet and Diaz, 1996] sur le formalisme CLP(FD). Un exemple est beaucoup plus explicite : on réécrit la contraint $3 * x < 5 + 5 * z - y$ sous la forme :

- $3 * x < 5 + 5 * MAX(z) - MIN(y)$
- $y < 5 + 5 * MAX(z) - 3 * MIN(x)$
- $5 * z > -5 + 3 * MIN(x) + MIN(y)$

En effet imposer la consistance des équations linéaires va seulement réduire le maximum ou le minimum des variables. On se place dans le pire cas pour ne pas perdre de solutions, d'où les expressions MAX(z) et MIN(y).

Une première étape de consistance peut ensuite se faire en temps $O(n^2)$ avec n le nombre de variables : on calcule l'expression, et on réduit le domaine de la variable en conséquence.

Il se passe quelque chose de différent ici par rapport aux autres algorithmes de consistance présentés : comme on donne un ordre arbitraire sur les variables, peut-être qu'une modification d'un domaine d'une des variables va entraîner la modification d'une autre (cette fois-ci avec la même contrainte). On parle de non idempotence de l'algorithme de consistance : il faut l'appliquer plusieurs fois pour assurer la consistance de la contrainte. Cela se fera lors de la propagation.

Équations linéaires :

De la même manière un exemple est plus adapté qu'un long discours. On prend la contrainte $z = 2x + 3y + 3$. Si $x \in \{-3; 1; 6\}, y \in \{-2; 3; 4\}$ alors $2x + 3y + 3 \in \{-9; -1; 6; 9; 14; 17; 24; 27\}$. Alors on sait que $z \in E$ avec $E = \{-9; -1; 6; 9; 14; 17; 24; 27\}$, donc on peut modifier le domaine de z de la manière suivante $D(z) \leftarrow D(z) \cap E$. On fait ça pour toutes les variables de la contrainte, et cela nous assure la consistance. Les deux cas sont :

- $z = \ldots$ alors $D(z) \leftarrow D(z) \cap E$
- $z \neq \ldots$ alors si $|E| = 1$, $D(z) \leftarrow D(z)/E$

Dans le deuxième cas, il faut que $|E| = 1$ car à nouveau on se place dans le cas le plus défavorable pour ne pas perdre de solutions.

La complexité reste assez élevée car l'addition de deux ensembles de taille m peut créer un ensemble de taille m*m a priori. En pratique cet algorithme est plus rapide que la recherche de supports car compte tenu de la non redondance dans les ensembles, la complexité pire cas sera rarement atteinte (par exemple $|[1;5]+[1;5]| = |[2;10]| = 9 \neq 25$)

All_different :

On a déjà présenté l'algorithme de consistance spécifique à cette contrainte globale. Pour arriver à avoir un programme qui fonctionne, il a fallu implémenter l'algorithme de recherche de couplage maximal de Hopcroft–Karp, l'algorithme de composantes fortement connexes (avec 2 parcours de graphe) et plusieurs parcours pour avoir la liste des valeurs inconsistantes. La représentation du graphe est faite par listes d'adjacence. Une première partie du pré-calcul est le fait de lister toutes les valeurs possibles que peuvent prendre chaque variable et créer le graphe.

La complexité est celle attendue dans pour trouver les valeurs inconsistantes ($O(n\sqrt{m})$). Pour la gestion incrémentale (quand on supprime une valeur) il y a seulement un facteur $log(n)$ en plus à cause de la structure de donnée utilisée pour lister les valeurs inconsistantes (on utilise des ensembles implémentés avec un arbre binaire de recherche fourni par Ocaml), cela pourrait s'améliorer avec des tables de Hachage avec une fonction de hachage convenable sur les couples de valeurs. La complexité est donc $O((n+m)log(n))$.

Autres :

Dans tous les autres cas (équations non linéaires, contraintes non globales), il faut quand même pouvoir résoudre. On applique donc la méthode de l'algorithme AC4 : on va faire une première recherche exhaustive de tous les supports, les stocker et compter pour chaque valeur combien elle a de supports. Cette première étape est très coûteuse, mais par la suite on pourra gagner du temps grâce aux supports stockés : lors de la suppression d'une valeur, on va regarder tous les supports dans lesquels elle intervenait, et les enlever de notre liste de supports. À ce moment, comme on sait combien de supports a chaque valeur, on sait si une valeur pour une variable est devenue inconsistante. Cela permet de propager la consistance.

La propagation est gérée de manière un peu plus globale pour pouvoir tenir compte de tous ces cas particuliers. C'est ce que nous allons voir maintenant.

3.3.2 Structure du code

Lors de l'exécution du programme, on donne en paramètre un problème de satisfaction de contraintes. Le parser et le lexer vont commencer par le donner sous une forme convenable. Ensuite ce problème est modifié pour tenir compte des cas particuliers : chaque contrainte est analysée et on lui donne un type (Eq_lin, All_dif, Other, ...). On créé aussi pour chaque variable les listes des contraintes qui font intervenir la variable.

La structure de données utilisée pour représenter les ensembles est un arbre binaire de recherche (fourni par une librairie d'Ocaml). Cela permet d'avoir les fonctions d'ajout, de suppression, de recherche de minimum et maximum en temps logarithmique.

Ensuite on entre dans la fonction de backtrack. Cette fonction va appeler la fonction qui rend le problème consistant, puis si on a une solution on l'affiche, sinon on choisit la variable qui a le plus petit domaine, et on teste récursivement toutes les affectations possibles de cette variable. Pour cela on copie toute la structure de donnée, cela est coûteux en mémoire mais cela évite de devoir recalculer des états. Néanmoins on ne va pas avoir trop de données en mémoire à chaque état car on n'a besoin de se souvenir que d'une structure de donnée par profondeur de l'arbre de recherche (donc pas plus que le nombre de variables).

La fonction de consistance va rendre le problème consistant par arcs. Elle est divisée en 2 parties : une partie propagation, et une partie qui analyse les contraintes pour trouver les valeurs à supprimer. Lors de la première consistance, on va initialiser toutes les structures de données des contraintes (graphe pour all_different, tableaux pour les contraintes `Other`). On appelle les consistances spécifiques à chaque contrainte lors de la propagation, et elles nous renvoient une liste de valeurs inconsistantes. Ensuite on choisit une valeur de cette liste, on la supprime, et on applique à nouveau les fonctions de consistances sur les contraintes qui sont peut-être devenues inconsistantes.

L'avantage de ce code est qu'il est très facile d'ajouter un type de contrainte avec une consistance spécifique : on ajoute le type dans le type somme, on crée la fonction de consistance associée, et le programme va appeler la bonne fonction de consistance dans ce cas. Avec ce code il est aussi

n-dames			
n	mode	temps (s)	nb_nœuds
5	fast	0.01	22
	slow	0.7	22
6	fast	0.02	67
	slow	0.7	63
7	fast	0.04	180
	slow	2.1	168
8	fast	0.18	663
	slow	X	X
9	fast	0.45	2574
10	fast	2	10071
11	fast	10	43420
12	fast	48	207037

FIGURE 4 – Temps de calcul pour le problème des n-dames

plus simple d'étendre aux domaines mixtes, car le code est très modulable (on s'est ramenés aux éléments de base de l'algorithme).

3.4 Quelques résultats

Il y a de nombreux problèmes qui peuvent se résoudre grâce à la programmation par contraintes, j'en ai choisi 2 pour tester mon implémentation et sa vitesse. Le but n'était pas de comparer avec les solveurs déjà existants qui sont dédiés aux entiers, et développés depuis des années, mais plutôt de voir si les améliorations faites accélèrent bien la résolution. Pour cela, j'ai effectué les tests sur des instances de plus en plus grandes avec et sans les améliorations. Sans les améliorations on va tout le temps rechercher les supports en évaluant l'expression, ce qui permet de traiter toutes les contraintes.

n-dames

Le problème des n-dames est un problème très connu : il s'agit de placer n-dames sur un échiquier de $n \times n$ sans qu'aucune ne s'attaque. À ce jour il n'existe aucune formule qui donne le nombre de combinaisons possibles de dames pour un n donné (de combien de façon on peut placer les dames). Il existe des algorithmes très rapides pour trouver *un seul* placement des dames, mais notre but ici est de trouver tous les placements. Les temps sont donnés en secondes dans le tableau 4. La représentation du problème se fait en écrivant les contraintes pour que les dames ne s'attaquent pas. Les variables x_i correspondent à la position de la dame sur la i-ème colonne. Le problème peut se représenter par les 3 contraintes :
all_different$(x_1, \ldots, x_n)$
all_different$(x_1 + 1, x_2 + 2, \ldots, x_n + n)$
all_different$(x_1 - 1, x_2 - 2, \ldots, x_n - n)$
L'implémentation des contraintes ne permettant pas d'avoir des expressions dans les paramètres de all_different, on a ré-écrit les 2 dernières contraintes avec $\forall i < j, x_i + i \neq x_j + j, x_i - i \neq x_j - j$. Le tableaux donne les temps en utilisant l'algorithme efficace de all_different (fast) et sans l'utiliser (slow), pour n dames. À partir de $n = 8$, sans l'algorithme efficace le temps de calcul est trop long (plus de 2 minutes). La colonne nb_noeuds donne le nombre d'appels à la fonction de backtrack (récursive).

Le nombre de noeuds ne varie pas beaucoup entre les deux algorithmes ce qui montre que l'étape qui prend du temps est la consistance de la contrainte `all_different`. Les différences de temps sont très élevées : on peut résoudre avec 4 dames de plus avec l'algorithme efficace.

Problème de Langford

Le problème de Langford est un autre problème combinatoire dont le temps de calcul explose très rapidement. On se donne deux fois chaque nombre entre 1 et n et le but est de générer une séquence où les deux 1 sont séparés par 1 autre nombre, les deux 2 sont séparés par 2 autres nombres, etc. Un exemple de solution pour $n = 4$ est "23421314". Une analyse peut montrer qu'il n'y a pas de

n-Langford			
n	mode	temps (s)	nb_noeuds
4	fast slow	0.01 0.01	10
5	fast slow	0.02 0.02	37
6	fast slow	0.04 0.08	128
7	fast slow	0.14 0.35	513
8	fast slow	0.68 2.1	2592
9	fast slow	3.8 15	14453
10	fast	23	81818
11	fast	2m34	495195

FIGURE 5 – Temps de calcul pour le problème de Langford

solution si $n \equiv 1, 2(mod 4)$, mais on peut toujours faire la recherche pour montrer qu'il n'y a aucune solution. Pour représenter le problème on utilise la contrainte all_different$(x_1, x_1 + 2, x_2, x_2 + 3, \ldots, x_n, x_n + n + 1)$. Cette fois-ci, le but est de tester l'efficacité des contraintes linéaires. On réécrit donc cette contrainte `all_different` en contraintes linéaires de différence. Ces contraintes ne feront intervenir que 2 variables chacune ce qui permet de se placer dans le cas le plus simple pour l'algorithme non amélioré : Si on prenait des contraintes linéaires avec beaucoup de variables, le temps de calcul exploserait pour l'algorithme non amélioré, alors qu'il est quadratique avec l'amélioration. On teste donc si dans le cas le plus défavorable, on est toujours meilleur. Le tableau des temps et nombre de noeuds est donné par la figure 5, mais à nouveau, sans l'algorithme pour la contrainte all_different, le temps de calcul est trop long à partir de $n = 10$. On voit qu'ajouter les contraintes linéaires améliore vraiment le temps de calcul (d'environ un facteur 4). Si on avait des contraintes linéaires d'arité plus grande (plus de variables) l'amélioration serait encore plus flagrante.

4 Conclusion

Finalement durant ce stage j'ai réussi à implémenter un solveur de contraintes discret, avec plusieurs fonctionnalités :

- Le code est modulable : on peut facilement ajouter des contraintes globales avec leur résolution spécifique. Pour l'instant il n'y a que les contraintes linéaires (égalité et inégalité) et `all_different`, mais il existe de nombreuses autres contraintes globales à traiter séparement, avec un algorithme spécifique
- La structure est aussi simple : j'ai séparé les parties importantes de la résolution ce qui fait qu'une heuristique peut être facilement implémentée (ajouter de l'aléatoire, un meilleur choix des variables dans le backtrack, un ordre pour traiter les contraintes, etc).

Ma présence à aussi permis d'améliorer le code du solveur continu Absolute déjà existant : j'ai trouvé quelques améliorations pour plus de généralité et pour pouvoir combiner les différentes méthodes de résolution.

Pendant tout mon stage j'ai cherché à avoir un code simple dans l'optique de pouvoir le combiner avec le solveur AbSolute qui ne permet, pour l'instant, de résoudre que des problèmes continus. Au fur et à mesure que mon code avançait, nous avons pu réfléchir à la méthode qui sera utilisée pour mixer les deux solveurs. Cela sera fait prochainement par les développeurs d'AbSolute (principalement Ghilles Ziat) avec qui j'ai travaillé.

Le but est de créer le tout premier solveur mixte, car aucun solveur actuel ne gère nativement les entiers et les réels en même temps. Ils vont seulement résoudre une partie et utiliser un autre solveur pour le reste, ou résoudre les entiers comme des réels et revenir aux entiers à la fin de la résolution.

Absolute va gérer nativement les deux types de résolution : d'un côté les réels avec leur algorithmes spécifiques, d'un autre les entiers avec des contraintes globales. On pourra ainsi dans une partie de pré-calcul séparer les contraintes en contraintes purement entières ou réelles et les contraintes mixtes, et effectuer la résolution en parallèle (en propageant les consistances entre les entiers et les réels).

Références

[Beldiceanu et al., 2012] Beldiceanu, N., Carlsson, M., and Rampon, J.-X. (2012). Global constraint catalog, (revision a).

[Bessiere, 1994] Bessiere, C. (1994). Arc-consistency and arc-consistency again. *Artificial intelligence*, 65(1) :179–190.

[Bessière et al., 2005] Bessière, C., Régin, J.-C., Yap, R. H., and Zhang, Y. (2005). An optimal coarse-grained arc consistency algorithm. *Artificial Intelligence*, 165(2) :165–185.

[Codognet and Diaz, 1996] Codognet, P. and Diaz, D. (1996). Compiling constraints in clp (fd). *The Journal of Logic Programming*, 27(3) :185–226.

[Cousot and Cousot, 1977] Cousot, P. and Cousot, R. (1977). Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM.

[Harvey, 1995] Harvey, W. D. (1995). *Nonsystematic backtracking search*. PhD thesis, stanford university.

[Mackworth, 1977] Mackworth, A. K. (1977). Consistency in networks of relations. *Artificial intelligence*, 8(1) :99–118.

[Mohr and Henderson, 1986] Mohr, R. and Henderson, T. C. (1986). Arc and path consistency revisited. *Artificial intelligence*, 28(2) :225–233.

[Pelleau et al., 2013] Pelleau, M., Miné, A., Truchet, C., and Benhamou, F. (2013). A constraint solver based on abstract domains. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 434–454. Springer.

[Régin, 1994] Régin, J.-C. (1994). A filtering algorithm for constraints of difference in csps. In *AAAI*, volume 94, pages 362–367.

[Rossi et al., 2006] Rossi, F., Van Beek, P., and Walsh, T. (2006). *Handbook of constraint programming*. Elsevier.

A Contexte du stage

Le stage s'est déroulé dans le laboratoire LS2N dans l'université de Nantes. L'équipe TASC dans laquelle j'étais est divisée en deux lieux : l'université de Nantes et les Mines de Nantes. À cause de cela et des dates du stage (fin d'année) je n'ai assisté qu'à une seule réunion d'équipe (qui s'est déroulée aux Mines de Nantes).

J'ai pendant toute la durée de mon stage partagé un bureau avec Ghilles Ziat qui est en thèse dirigée par Antoine Miné et Charlotte Truchet, et qui est un des développeurs principaux d'AbSolute. Je le remercie pour toute l'aide qu'il m'a fourni pour la compréhension du code et les habitudes de programmation sur le langage Ocaml.

J'ai aussi pu parler avec les nombreux doctorants du laboratoires ce qui m'a permi de mieux comprendre les enjeux de la thèse, et de découvrir d'autres domaines de l'informatique dont je n'avais jamais entendu parler.

Lors du stage je suis allé à Rennes pour suivre une explication du fonctionnement d'Absolute dans le laboratoire de l'INRIA. Cela a été fait car à la rentrée, il y aura un doctorant qui travaillera aussi sur le solveur, pour l'implémentation des domaines paramètrés (des intervalles de la forme $[1; n]$ par exemple, avec n qui est un paramètre et non une variable).

Ce stage m'a donné une très bonne impression du monde de la recherche, et m'a vraiment plu, tant sur le sujet que sur l'ambiance qui régnait dans le laboratoire.