Faculté des sciences et génie
Département d'informatique et de génie logiciel

IFT-7020 - Optimisation Combinatoire

Automne 2024

Implémentation d'un algorithme de filtrage de contrainte globale à l'aide de nombres duaux

Bourez Nicolas — **IDUL**: 537315033
Onciul Alexandra — 537305977
Miralles Alexandre — 537316065

Professeur:
Quimper Claude-Guy

December, 2024

1 Introduction

L'optimisation combinatoire est une branche des mathématiques qui vise à résoudre une multitude de problèmes liés à la recherche de la meilleure solution parmi un ensemble fini ou infini d'options possibles, sous des contraintes spécifiques. Ces problèmes apparaissent fréquemment dans des domaines tels que la gestion des ressources, la planification, la logistique, la conception de réseaux, ainsi que dans des applications pratiques comme l'allocation des tâches, le routage, ou encore la recherche de configurations optimales dans des systèmes complexes.

L'objectif est de trouver des solutions optimales ou proches de l'optimal, en tenant compte de critères divers, souvent sous forme de fonctions objectif à maximiser ou minimiser.

Cependant, une grande majorité de ces fonctions requiert des algorithmes de filtrage spécifique à la fonction, et nécessite parfois un développement mathématique très compliqué et parfois incompréhensible pour les personnes extérieures au domaine. Parmi ces problèmes, la minimisation de fonctions de dispersion telles que la variance et l'index de Gini ont fait naître des algorithmes de filtrage très complexes.

Dans cet article, on envisage l'implémentation d'un algorithme de filtrage global, à qui l'on pourrait donner n'importe quelle fonction objectif et qui nous donnerait, potentiellement la meilleure, une solution ou du moins une borne satisfaisante pour la plupart des applications. Cet algorithme serait basé sur un concept: exploiter l'arithmétique des intervalles et la monotonicité des fonctions, comme le fait [Araya et al. 2010], mais en l'adaptant pour des contraintes globales plutôt que des contraintes arithmétiques. En plus de cela, nous exploiterons les nombres duaux afin d'obtenir une différentiation exacte et simple des fonctions. Nous finirons par appliquer notre algorithme à une application concrète, à savoir sur un problème de minimisation de dispersion des variables. Pour cela, nous appliquerons notre algorithme à la fonction `spread` et nous comparerons les résultats obtenus avec les résultats existants dans la littérature.

2 Notations

Définition 1 Un intervalle flottant fermé/ouvert est un ensemble connexe de réels dont la borne supérieure et la borne inférieure sont des nombres à virgule flottante. Les notations suivantes sont utilisées comme abréviations : $[g..h] \equiv \{r \in \mathbb{R} | g \leq r \leq h\}, [g..h) \equiv \{r \in \mathbb{R} | g \leq r < h\}$. Par défaut, $\mathbb{I}$ est l'ensemble des intervalles reéls tandis que $\mathbb{IZ}$ est l'ensemble des intervalles entiers.

L'arithmétique des intervalles a été définie pour étendre aux intervalles $\mathbb{I}$ les fonctions élémentaires définies sur $\mathbb{R}$. Lorsque une fonction f est une composition de fonctions élémentaires, une extension de f aux intervalles doit être définie afin de garantir un calcul conservatif de l'image.

Définition 2 Une extension en intervalle de $f : \mathbb{R}^n \to \mathbb{R}$ est une application $F : \mathbb{I}^n \to \mathbb{I}$ telle que, pour tous $I_1, \ldots, I_n \in \mathbb{I}$:

$$r_1 \in I_1, \ldots, r_n \in I_n \implies f(r_1, \ldots, r_n) \in F(I_1, \ldots, I_n).$$

Définition 3 Un problème de satisfaction de contraintes numériques est un problème $P = (X, D, C, o)$ avec $X = [X_1, \ldots, X_n]$ un ensemble de n variables réelles, D une fonction qui associe à chaque variable $x \in X$ un domaine tel que que $x \mapsto D(x) \subseteq \mathbb{I}$, un ensemble de contraintes C sur les variables X et une fonction objectif o à minimiser/maximiser tel que $o : X \mapsto o(X) : \mathbb{R}^n \to \mathbb{R}$. $\rho_c \subseteq \mathbb{R}^n$ est la relation inhérente à une contrainte $c \in C$. L'ensemble des solutions du problèmes est donné par l'ensemble $S = \bigcup_{c \in C} \rho_c$. La solution s est optimale si $o(s) \leq o(s') \forall s' \in S \wedge s \neq s'$ (resp. $\geq$ pour un problème de maximisation). On définit $\underline{X_i} = \min(\text{dom}(X_i))$ et $\overline{X_i} = \max(\text{dom}(X_i))$.

Note Le but de notre algorithme est de borner les valeurs de la fonction objectif. Dans ce contexte, nous pouvons définir $V = o(X)$ comme la variable correspondant à la fonction objectif. Cette variable possède initialement un domaine $\text{dom}(V) = (-\infty..\infty) \in \mathbb{I}$. Supposons que les domaines de nos variables X soient entiers, cela n'affectera pas la cohérence de borne de notre variable V puisque celle-ci a un domaine de variables réelles.

3 Notions utilisées dans l'algorithme

3.1 Les nombres duaux

L'ensemble des nombres duaux, noté $\mathbb{D}$, est une extension de l'ensemble des réels $\mathbb{R}$, d'une manière analogue aux nombres complexes [Agarwal et al. 2023]:

- Là où les nombres complexes augmentent les réels en introduisant un nombre *imaginaire* $i^2 = 1$
- Les nombres duaux augmentent l'ensemble des réels en introduisant un nombre *infinitésimal* $\epsilon^2 = 0$ et $\epsilon \neq 0$.

Un nombre dual prend donc la forme:

$$z = a + v\epsilon$$

Avec la composante réelle a et la composante infinitésimal v. Cette simple augmentation des nombres réels permet de faire émerger une méthode simple pour calculer des dérivés *exactes* sans pour autant avoir besoin de manipuler des expressions mathématiques complexes.

Pour pouvoir utiliser les nombres duaux dans le calcul de dérivée, on définit la valeur a comme étant la **valeur primale**, à savoir l'input initial de la fonction, et la valeur v comme étant la dérivée de cette valeur primale a, qu'on note $\dot{a}$ et qu'on appelle **valeur tangente** [Baydin et al. 2018]µ. On obtient dès lors:

$$z = a + \dot{a}\epsilon$$

La magie s'opère lorsqu'on s'intéresse au développement de Taylor d'une fonction quelconque évalue en ce nombre:

$$f(z) = f(a + \dot{a} * \epsilon) = \sum_{i}^{\infty} \frac{f^{(n)}(a) * \dot{a}^n * \epsilon^n}{n!}$$

Puisque $e^2 = 0, e^3 = e^2 * e = 0, ...$, le développement de Taylor se simplifie en:

$$f(a + v\epsilon) = f(a) + f'(a) * \dot{a} * \epsilon \tag{1}$$

En conclusion, ajouter une partie infinitésimale $\dot{a} * \epsilon$ à la valeur primale a permet de faire apparaître la valeur de la dérivée $f'(a)$ en cette valeur.

Exemple:

$$\begin{aligned} f(x) &= 3x^2 \\ f(x + \dot{x} * \epsilon) &= 3(x + \underbrace{\dot{x}}_{1} * \epsilon)^2 \\ &= 3(x^2 + 2x\epsilon + \epsilon^2) \\ &= 3x^2 + 6x\epsilon + 3\epsilon^2 \\ &= f(x) + \epsilon f'(x) \end{aligned}$$

$$\Rightarrow \quad f'(x) = 6x$$

Enfin, si on applique la règle de la chaine sur l'equation 1, on obtient:

$$\begin{aligned} f(g(a + \dot{a}\epsilon)) &= f(g(a)) + g'(a)\dot{a}\epsilon) \\ &= f(g(a)) + f'(g(a))g'(a)\dot{a}\epsilon. \end{aligned}$$

Le coefficient de ϵ dans le membre de droite correspond exactement à la dérivée de la composition de f et g. Cela signifie que, puisque nous implémentons des opérations élémentaires de manière à respecter l'équation 1, toutes leurs compositions respecteront également le pattern donné par l'équation.

3.1.1 Opérations sur les nombres duaux

On peut définir un nombre dual à l'aide d'un tuple ($\texttt{var}, \texttt{derivée}$). Par exemple, si on définit deux nombres duaux:

$$u = (x, \dot{x}) = x + \dot{x}\epsilon$$
$$v = (y, \dot{y}) = y + \dot{y}\epsilon$$

On peut définir des opérations sur ces nombres:

$$\begin{aligned} \texttt{add}(u, v) = &\quad (x + y, \dot{x} + \dot{y}) \\ \texttt{sub}(u, v) = &\quad (x * y, x * \dot{x} + y * \dot{y}) \\ \texttt{mul}(u, v) = &\quad (x - y, \dot{x} - \dot{y}) \\ \texttt{div}(u, v) = &\quad \left(\frac{x}{y}, \frac{x*\dot{x} - y*\dot{y}}{\dot{y}^2}\right) \end{aligned}$$

On remarque que la valeur du deuxième terme correspond à chaque fois à la dérivée de l'opération entre les premières composantes x et y. Dans un programme informatique, les nombres duaux peuvent donc être utilisée comme structure de donnée qui sont capables de supporter deux informations: la valeur primale et la valeur tangente de l'input d'une fonction.

3.1.2 Utilisation des nombres duaux

L'égalité 1 permet l'interprétation suivante:

Nous pouvons extraire la dérivée d'une fonction en interprétant tout nombre non-dual v comme $v + 0\epsilon$ et en évaluant la fonction de cette manière non standard sur une entrée initiale avec un coefficient de 1 *pour* ϵ :

$$\left.\frac{df(x)}{dx}\right|_{x=v} = \text{coefficient de } \epsilon\big(\text{version-duale}(f)(v + 1\epsilon)\big) \tag{2}$$

Cette propriété est particulièrement utilisée dans le contexte de la différentiation automatique. C'est une technique utilisée pour calculer les dérivées en décomposant des fonctions en opérations élémentaires dont les dérivées sont bien connues. L'un des modes de cette différentiation, le mode direct ou *forward*, utilise cette propriété pour calculer les dérivées sans avoir besoin de dérivation symbolique ou d'approximation numérique (comme les différences finies). Chaque opération élémentaire étant étendue pour respecter l'équation 2, toute fonction composée d'opérations élémentaires respectera également cette propriété de manière automatique.

3.2 La monotonicité des fonctions et son lien avec l'arithmétique d'intervalle

Comme dans l'article de [Araya et al. 2010], nous décidons d'exploiter l'arithmétique d'intervalle et la monotonicité de certaines fonctions pour pouvoir borner celles-ci.

Définition 4 Une fonction $f(x_1, \ldots, x_n)$ est **croissante** par rapport à x_i sur les intervalles $[a_j, b_j]$ si pour n'importe quelles valeurs de $x_1, \ldots, x_n$ avec $x_j \in [a_j, b_j]$, nous avons la relation suivante :

$$\frac{\partial}{\partial x_i} f(x_1, \ldots, x_n) \geq 0 \quad \forall x_j \in [a_j, b_j]$$

Définition 5 Une fonction $f(x_1, \ldots, x_n)$ est **décroissante** par rapport à x_i sur les intervalles $[a_j, b_j]$ si pour n'importe quelles valeurs de $x_1, \ldots, x_n$ avec $x_j \in [a_j, b_j]$, nous avons la relation suivante :

$$\frac{\partial}{\partial x_i} f(x_1, \ldots, x_n) \leq 0 \quad \forall x_j \in [a_j, b_j]$$

Définition 6 ($F_{\min}, F_{\max}$, extension basée sur la monotonicité)
Soit f une fonction définie sur les variables X_i, et F son extension aux intervalles. Soit $M \subseteq X$ un sous ensemble de variables monotones.

On scinde les variables en deux groupes:

- les variables M_i^+ qui sont des variables monotones croissantes
- les variables M_i^- qui sont des variables monotones décroissantes

On considère les variables $N = X \setminus M$ l'ensemble des variables non-monotones. On définit les fonctions $F_{\min}$ et $F_{\max}$ comme:

$$F_{\min}(W) = f(\underline{M_1}^-, \ldots, \underline{M_n}^-, W)$$

$$F_{\max}(W) = f(\overline{M_1}^+, \ldots, \overline{M_n}^+, W)$$

Finalement, l'extension basée sur la monotonicité F_{mono} de F sur les domaines des variables produit l'interval image suivant:

$$F_{mono}(\mathrm{dom}(X_1), \ldots, \mathrm{dom}(X_n)) = \left[\underline{F_{\min}([W])}, \overline{F_{\max}([W])}\right]$$

Propriété 1 Lorsque les variables n'apparaissent qu'une seule fois dans une fonction, l'arithmétique d'intervalles mène à des bornes exactes

Cependant, comme l'explique [Araya et al. 2010], ce n'est plus le cas si une variable est présente plusieurs fois dans la fonction à évaluer. On appelle cela le *problème de dépendance*. En effet, trouver l'intervalle exact d'un polynôme est NP-Hard [Kreinovich et al. 2013]. Cependant, les fonctions monotones permettent de trouver un intervalle exact même dans le cas de multiple occurences.

Propriété 2 Soit une fonction $F : B \mapsto b : \mathbb{I}^n \to \mathbb{I}$ sur un vecteur d'intervalles $B = (\mathrm{dom}(X_i), ..., \mathrm{dom}(X_n))$ alors:

$$F_{opt}(B) \subseteq F_{mono}(B) \subseteq F(B)$$

Si une fonction f est monotone par rapport à toutes ses variables (même celles présentent plusieurs fois), alors appliquer l'arithmétique des intervalles sur cette fonction permet de trouver un intervalle exact et optimal:

$$F_{opt}(B) = F_{mono}(B)$$

Appliquer l'arithmétique d'intervalle sur des fonctions non-monotones aura malheureusement pour effet de donner un surensemble de valeurs retournées par la fonction. Si certaines variables sont monotones, mais pas toutes, appliquer F_{mono} permettra une meilleure approximation.

4 Description de l'algorithme

L'algorithme que nous allons présenter dans cette section est une version simpliste de l'algorithme MOHC (MOnotonic Hull Consistency) introduit par [Araya et al. 2010].

L'algorithme prend en argument trois éléments: une fonction $F : B \mapsto b : \mathbb{I}^n \to \mathbb{I}$, un vecteur de variables X ainsi que qu'un vecteur B contenant les domaines de nos différentes variables. La première étape consiste à instancier une variable booléenne `bool` à la valeur en sortie de la fonction `CheckMultiplicity(F,X)`. Celle-ci va retourner `True` si au minimum une des variables de la fonction apparaît plusieurs fois dans celle-ci, `False` sinon. Si aucune variable n'est présente plusieurs fois, il nous suffit d'appliquer l'arithmétique d'intervalle à notre fonction et on aura la solution optimale. Sinon, on rentre la condition `then`.

La fonction `CheckMonotonicity(F,B)` prend en argument la fonction F et le vecteur de domaines B, et va vérifier la monotonicité de la fonction par rapport à ses différentes variables à l'aide des nombres duaux.

Algorithm 1 BoundingFunction (F, X, B)

```
// Initialisation
bool ←CheckMultiplicity(F,X)
lower_bound, upper_bound ← (−∞, ∞)
if bool then
    (M⁻, M⁺, W) ← CheckMonotonicity(F,B)
    lower_bound, upper_bound ←evaluateFunctionWithBounds(B,(M⁻, M⁺, W), F)
else
    lower_bound, upper_bound ← F(D)
end if
return lower_bound, upper_bound
```

Pour cela, nous appliquons les nombres duaux et vérifions si les dérivés partielles sont du même signes. Pour les variables croissantes, nous gardons les mêmes bornes lors de l'évaluation, pour celles décroissante, nous échangeons les bornes afin de garantir l'arithmétique des intervals. Pour les variables non-monotoniques, nous gardons les bornes telle quelles sont mais, sans garantie, qu'il s'agit de l'intervalle le plus serré. C'est la fonction **evaluateFunctionWithBounds** qui s'en occupe.

Enfin, l'algorithme retourne deux valeurs, à savoir des bornes inférieures et supérieures pour notre fonction en entrée.

Nous avons décidé de l'implémenter en Java afin de pouvoir l'intégrer a MiniCP qui est un solveur accessible quant à sa complexité. Nous allons décrires les classes principales de notre implémentation.

- **TraductMathematical expression**: Il s'agit d'une classe permettant de parcourir un string représentant une fonction et de calculer le résultat de cette fonction avec un nombre dual. Elle utilise notre classe **Regex** afin de pouvoir prendre en compte des fonctions personnalisées.
- **Dual**: Une classe d'évaluer les dérivées partielles des bornes de manière rapide de n'importe quelle expression basique (composée de d'additions, de soustraction, de multiplication, de division et de parenthèse). Afin d'y parvenir, nous y avons intégré l'arithmétique des intervalles directement dans cette classe car appliquer les dérivées partielles seulement aux bornes positives et négatives fausserait notre résultat.
- **Monotonicity**: Elle vérifie la monotonicité et retourne les bornes de la fonction.

5 Protocole d'expérimentation

Afin d'évaluer notre algorithme, nous avons décidé d'appliquer la fonction *variance* définie comme:

$$F(X) = \sigma_X^2 = \frac{1}{n}\sum(x_i - \mu)^2 \text{ ou } F(X) = \frac{1}{n}\sum X_i^2 - \mu^2 \tag{3}$$

avec μ la moyenne arithmétique définie comme $\mu = \frac{1}{n}\sum x_i^2$. Si on développe l'expression, on a:

$$\begin{aligned} F(X) &= \frac{1}{n}\sum X_i^2 - \left(\frac{1}{n}\sum X_i\right)^2 \\ &= \frac{1}{n}\sum X_i^2 - \frac{1}{n^2}\left(\sum X_i^2 + 2\sum_{1\le i\le j\le n} x_i x_j\right) \end{aligned}$$

Ce qu'on peut remarquer au préalable, c'est que , mis à part lors du cas $n = 2$ où l'expression est factorisable, l'expression fait apparaître une multiplicité > 1 pour chacune des variables. Il sera donc intéressant de voir comment l'étude de la monotonicité va influencer les résultats.

Cette fonction a été beaucoup étudiée au cours de ces dernières années, notamment dans le but de minimiser celle-ci. En effet, dans certains domaines, notamment l'agencement d'horaires, la dispersion est une

mesure importante car on aimerait une répartition équitable entre les employés. Cela se traduit par une minimisation de la variance dans un contexte d'optimisation combinatoire.

Afin de juger l'efficacité de notre algorithme à filtrer les valeurs que peuvent prendre cette fonction, nous utiliserons la méthode utilisée dans le récent article de [Ek et al. 2022].

Celui-ci introduit un algorithme de filtrage de la variance et de l'index de gini pour une moyenne non fixée au préalable. Celui-ci est donc plus complet que le nôtre car il permet de filtrer également les variables du problèmes, mais il se contente que de borner inférieurement la variance.

Nous allons donc comparer nos résultats à ceux fournit par:

- Une décomposition de la variance dans MiniZinc

```
predicate spread(array[int] of var int:X, var int:M, var int:v, int:s) =
    let { int: n = length(X);
          array[int] of var 0..infinity: sq = [x * x | x in X];
          var 0..n*ub(sum(sq)): v_sq = n*sum(sq);
          var 0..max(lb(M)*lb(M), ub(M)*ub(M)): Msq = M*M;
          var 0..ub(v_sq): v__ = v - Msq;
          var 0..ub(v__)*s: v__ = v__*s; }
    in v = v__ div (n*n) /\ M = sum(X);
```

- L'algorithme de la contrainte `variance` présenté dans [Araya et al. 2010], utilisable via le solveur `Chuffed`.

Nous avons alors implémenté leur filtrage en Java afin de savoir si notre implémentation filtre plus ou moins que celle de MiniZinc, son implémentation est dans **SpreadBoundsReduction** pour différents scénarios.

Nous nous sommes basé sur la méthode de test de [Araya et al. 2010] mais avec un nombre de variables plus restreint étant donné que notre traduction de string à fonction est limitée, nous devons développer les exposants de la variance ce qui nous limite dans notre protocole d'expérimentation.

Voici le procédé, nous prenons comme fonction à filtrer, la variance pour $n \in 2, 3, 4, 5$. Pour choisir les bornes inférieures et supérieures, nous appliquons une fonction random entre -50 et 50, si les deux nombres sont identiques, nous en piochons d'autres. Pour chaque n, nous faisons 5 instances ce qui nous donne un total de 20 instances.

6 Résultat

Nous avons, tout d'abord, une classe **TestMonotonicity** qui reprend 2 exemples vus au cours afin de tester si tout fonctionne bien ensemble.

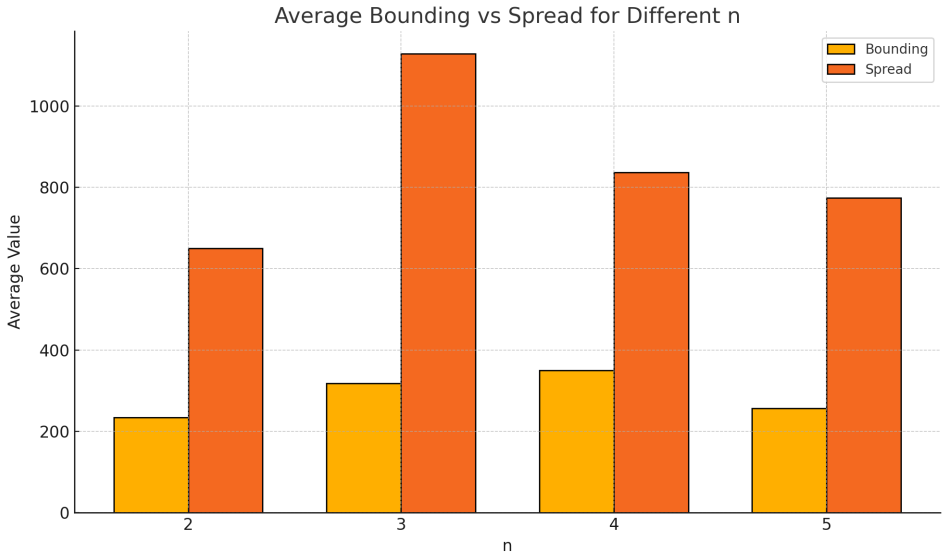
```
Bounds using Monotonicity.boundingFunction on : X1*X1 - X1*Y1*Y1
Lower Bound: 5.0
Upper Bound: 272.0
```

```
Bounds using Monotonicity.boundingFunction on : 3*X - X*Y + 4*Y - 12
Lower Bound: 0.0
Upper Bound: 15.0
```

Dans la classe **CompareVariance**, vous trouverez le protocole d'expérimentation. La méthode spread génère des intervalles plus larges de manière globale. Notre méthode (Bounding), en revanche, fournit des bornes plus resserrées et peut donc être mieux adaptée pour le filtrage de domaine. Nous pouvons observer qu'il y a une différence considérable entre les deux filtrages.

7 Discussion

Afin de pouvoir tester plus fidèlement notre implémentation, il faudrait implémenter ce filtrage dans un solveur et voir comment il se comporte. MiniCP serait une bonne alternative.



Ensuite, l'ajout de plus de possibilités de traductions de fonctions comme les exposants ou encore les exponentielles pourrait être bénéfique, dans notre implémentation seulement les opérations basiques sont disponibles, car l'ajout d'autres fonctions à détecter augmentait les possibles erreurs de logique lors de la phase de traduction, ceci du à la complexité du code.

Nous voulons également souligner une complexité accrue dans les cas où des variables multiples influencent la fonction de manière non monotone.

Nous soupçonnons également que des erreurs se soient glissées, possiblement dans notre traduction de string à fonction notamment, étant donné la performance remarquable de notre implémentation que ce soit au niveau de notre implémentation du filtrage ou bien de l'implémentation de la contrainte Spread en Java.

8 Conclusion

Ce projet a démontré l'efficacité de l'algorithme proposé pour le filtrage de la variance dans des contextes de contraintes globales. Les résultats montrent que notre méthode Bounding génère des bornes plus serrées que celles obtenues avec la méthode Spread, ce qui la rend plus adaptée pour des scénarios nécessitant un filtrage précis des domaines.

En somme, ce travail constitue une avancée dans l'utilisation des nombres duaux et de la monotonicité pour améliorer le filtrage d'intervalles dans les problèmes d'optimisation combinatoire, tout en ouvrant la voie à des recherches futures pour une validation et une application plus étendue.

References

[SW] Sameer Agarwal, Keir Mierle, and The Ceres Solver Team, *Ceres Solver* version 2.2, Oct. 2023. URL: `https://github.com/ceres-solver/ceres-solver`.

Ignacio Araya, Gilles Trombettoni, and Bertrand Neveu. 2010. "Exploiting monotonicity in interval constraint propagation". In: *Proceedings of the AAAI Conference on Artificial Intelligence* 1. Vol. 24, 9–14.

Atilim Gunes Baydin, Barak A Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. 2018. "Automatic differentiation in machine learning: a survey". *Journal of machine learning research*, 18, 153, 1–43.

Alexander Ek, Andreas Schutt, Peter J Stuckey, and Guido Tack. 2022. "Explaining Propagation for Gini and Spread with Variable Mean". In: *28th International Conference on Principles and Practice of Constraint Programming (CP 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

Vladik Kreinovich, Anatoly V Lakeyev, Jiří Rohn, and PT Kahl. 2013. *Computational complexity and feasibility of data processing and interval computations*. Vol. 10. Springer Science & Business Media.