Résolution des CSPs continus

Julien Alexandre dit Sandretto

Department U2IS

ENSTA Paris

IA302-2020-2021

Adaptation des algorithmes du discret

Deux grands principes pour la résolution des CSPs discrets :

- l'algorithme systématique (avec sa variante de simple retour arrière)
- le filtrage en fonction de la consistance

Filtrage

Dans le domaine continu : filtrage = contracteurs

Contracter un CSP consiste à remplacer une boîte $[x]$ par une boîte plus petite $[x']$, telle que l'ensemble solution reste inchangé. C'est à dire, si $\mathcal{S} \subset [x]$, alors $\mathcal{S} \subset [x'] \subset [x]$.

Les propriétés d'un contracteur sont :

- Contractance : $\forall [x], \quad C([x]) \subset [x]$
- Correction : $\forall [x], \quad [x] \cap \mathcal{S} \subset C([x])$

Les contracteurs peuvent être combinés $C_1(C_2([x]))$ ou embarqués $C_1(C_2, [x])$

Contracteurs

Les contracteurs les plus connus :

- $Ctc_{fwd/bwd}([x])$, reposant sur la programmation par contrainte (le Forward/Backward)
- $Ctc_N([x])$, un algorithme de Newton (pour les problèmes carrés), de plus, il prouve l'existence et l'unicité de la solution
- $Ctc_{FixedPoint}(Ctc, [x])$, un contracteur de point fixe. Il appelle itérativement le contracteur Ctc
- $Ctc_{cid}(Ctc, [x])$, réalise des tranches de $[x]$, appelle Ctc sur chaque tranche, et retourne l'union des résultats.

Quelques propriétés

Un contracteur Ctc est :

- Monotone si et seulement si $[p] \subset [q] \Rightarrow Ctc([p]) \subset Ctc([q])$
- Minimal ssi $\forall [p] \in \mathbb{IR}^n, Ctc([p]) = [[p] \cap S]$
- Fin ssi $\forall p \in \mathbb{R}^n, Ctc(p) = \{p\} \cap S$
- Idempotent ssi $\forall [p] \in \mathbb{IR}^n, Ctc(Ctc([p])) = Ctc([p])$
- plus contractant que Ctc' ssi $\forall [p] \in \mathbb{IR}^n, Ctc([p]) \subset Ctc'([p])$

Méthode de branchement

Algorithmes systématiques du monde discret : génèrer une affectation totale ou partielle

On choisie une valeurs parmi celles possibles dans le domaine et on l'assigne à la variable.

Dans le continu, on ne va évidement pas piocher toutes les valeurs possibles, puisqu'il y en a une infinité

Méthode de branchement

Considérons un domaine continu : $[x]$, alors deux affectations possibles, sans perte de solution, sont $[x_1]$ et $[x_2]$ tels que $[x_1] \cup [x_2] = [x]$.

Cette découpe d'un intervalle en deux intervalles telle que toutes les valeurs possibles sont conservées = **bissection**.

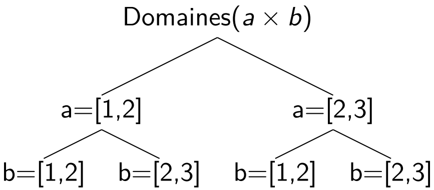
La bissection peut se réaliser au milieu : $[x] = [\underline{x}, m(x)] \cup [m(x), \overline{x}]$ ou à 90% par exemple

Boîte (un vecteur d'intervalles) : choisir la dimension à bissecter (rappel des heuristiques du deuxième cours)

La bissection $\Rightarrow$ le pavage (intérieur ou extérieur) d'un ensemble, ou un arbre de recherche dans le domain initial

Méthode de branchement

Par exemple, parcourir le domaine donné par une boîte $[1,3] \times [1,3]$, après deux bissections (une sur la première dimension et une sur la deuxième) :



Heuristiques de bissection

- Choix de la variable à bissecter (LargestFirst, RoundRobin)
- Choix du nombre (pas forcément en 2) ou de l'endroit de la bissection (pas forcément au milieu)

Mais aussi les approches rognage (ou slicer) :
découper les bords d'un intervalle et essayer de l'invalider pour l'enlever (3bCid)

Les algorithmes classiques

Deux algorithmes de branchement les plus utilisés (dans leur version simple) :

- Branch & Prune
- Branch & Contract

Branch & Prune

Traduction directe du génère et teste au domaines continus

Calculer un pavage approximant l'ensemble solution (si il n'est pas vide)

2 types de contraintes sont nécessaires :

- validant un domaine C_{in}, c'est à dire faisant partie de l'ensemble solution et donc une boîte intérieure
- invalidant un domaine C_{out}, c'est à dire dont aucun point ne fait partie de l'ensemble solution et donc une boîte extérieure

Branch & Prune

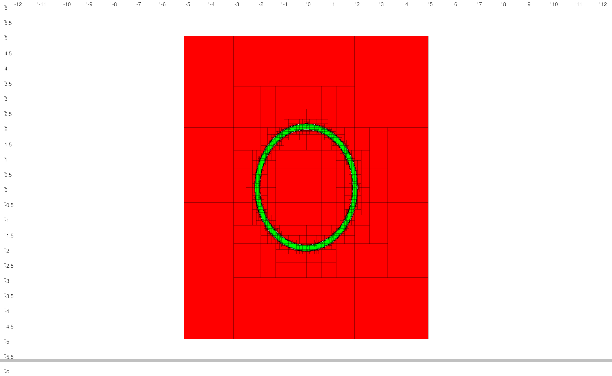
Require: $Stack = \emptyset, Stack_{acc} = \emptyset, Stack_{rej} = \emptyset, Stack_{unc} = \emptyset, [X]_0 \subset \mathbb{R}^{\cdot\cdot}$
  Push $[X]_0$ in $Stack$
  while $Stack \neq \emptyset$ **do**
    Pop a $[X]$ from $Stack$
    if $C_{in}([X])$ **then**
      Push $[X]$ in $Stack_{acc}$
    else if $C_{out}([X])$ **then**
      Push $[X]$ in $Stack_{rej}$
    else if $\text{width}([X]) > \tau$ **then**
      $([X_{left}], [X_{right}]) = \text{Bisect}([X])$
      Push $[X_{left}]$ in $Stack$
      Push $[X_{right}]$ in $Stack$
    else
      Push $[X]$ in $Stack_{unc}$
    end if
  end while

Exemple Branch & Prune

Calcul des points faisant partie d'un anneau centré en zéro et dont le rayon est compris entre 1,9 et 2,1

- $\mathcal{X} = \{x, y\}$
- $\mathcal{D} = \{[-5, 5]^2\}$
- $\mathcal{C} = \{x^2 + y^2 \in [1.9, 2.1]\}$

Contrainte de rejet : $x^2 + y^2 \notin [1.9, 2.1]$



Branch & Contract

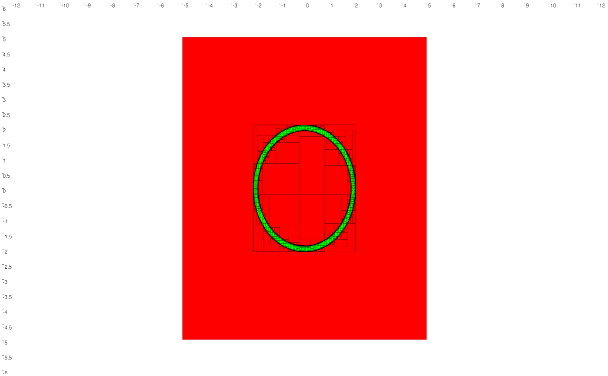
Utilise le filtrage pour accélérer la résolution

Branch & Contract

Require: $Stack = \emptyset, Stack_{acc} = \emptyset, Stack_{rej} = \emptyset, Stack_{unc} = \emptyset, [X]_0 \subset \mathbb{R}$

Push $[X]_0$ in $Stack$
while $Stack \neq \emptyset$ **do**
  Pop a $[X]$ from $Stack$
  Contract $[x]$ w.r.t. C_{in}
  if $C_{in}([X])$ **then**
    Push $[X]$ in $Stack_{acc}$
  else if $C_{out}([X])$ **then**
    Push $[X]$ in $Stack_{rej}$
  else if width$([X]) > \tau$ **then**
    $([X_{left}], [X_{right}]) = \text{Bisect}([X])$
    Push $[X_{left}]$ in $Stack$
    Push $[X_{right}]$ in $Stack$
  else
    Push $[X]$ in $Stack_{unc}$
  end if
end while

Exemple Branch & Contract



Comparaison arbres de recherche

Effet de la contraction :

- le nombre bien inférieur de bissections nécessaires (plus rapide)
- plus petit arbre de recherche (moins de mémoire)
- une précision supérieure pour un seuil de bissection similaire (plus précis)

Fonctionnalités avancées Ibex

```
Variable x(2);
Function func(x, x[0]+x[1]);
//ou Function func(x,Return(x[1], 2*x[0]));
NumConstraint cst(func, EQ); ou LEQ
CtcFwdBwd ctc(cst);

IntervalVector box(2,Interval(0,1));
ctc.contract(box);

LargestFirst bb(0.01);
pair<IntervalVector, IntervalVector>
              box_bis = bb.bisect(box);

IntervalVector box_first=box_bis.first;
ctc.contract(box_first);
```

TP : écrire un solveur de CSP continus

En utilisant Ibex, programmer un solveur de CSP (idéalement un Branch & Contract) capable de caractériser la solution au problème suivant :

Quels sont les points de $[-3, 3]^2 \subset \mathbb{R}^2$ qui sont à la fois définis comme étant dans l'image inverse de $[-0.1, 0.1]$ par la fonction $f(x, y) = x^4 - x^2 + 4y^2$ et à une distance de $[0.9, 1.1]$ du zéro.

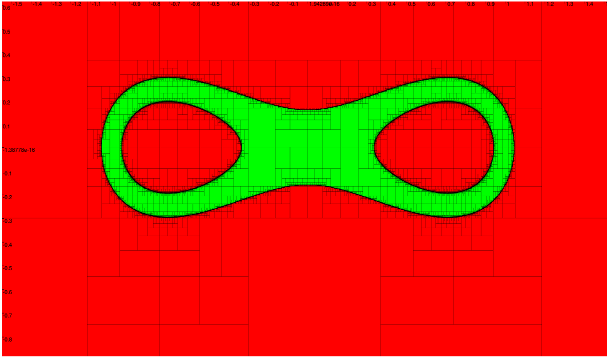
Essayez différentes heuristiques pour améliorer la rapidité/qualité.

Solution

Voir fichier cpp

Solution

Image de $f(x, y) = x^4 - x^2 + 4y^2$:



Solution

Intersection avec cercle :