Chapitre 1

Introduction aux CSP

Sommaire

Ce chapitre présente, tout d'abord, les notions de base relatives aux Problèmes de Satisfaction de Contraintes (CSP) : cohérence, filtrage et méthodes de recherche. Puis, il s'intéresse aux contraintes réifiées qui seront utilisées pour modéliser les problèmes d'extraction de motifs ensemblistes.

1.1 Définitions

Définition 1.1. CSP
Un CSP est un triplet $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ tel que :

- $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$ est l'ensemble des **variables**,
- $\mathcal{D} = \{D_1, D_2, \ldots, D_n\}$ est l'ensemble des **domaines des variables**. Chaque domaine D_i est un ensemble fini contenant les valeurs possibles pour la variable x_i.
- - $\mathcal{C} = \{c_1, c_2, \ldots, c_e\}$ est l'ensemble des **contraintes**.

Chaque contrainte c_i porte sur un sous-ensemble de variables de $\mathcal{X}$ appelé $var(c_i)$. L'arité (nombre de variables) de c_i est définie par $|var(c_i)|$.

Définition 1.2. Instanciation
Une **instanciation élémentaire** I est un couple variable-valeur (x_i, v) tel que $v \in D_i$, qui est le domaine de x_i. Une **instanciation** est un ensemble d'instanciations élémentaires. Si une instanciation porte sur toutes les variables alors on parle d'**instanciation complète**, sinon il s'agit d'**instanciation partielle**.

Définition 1.3. Solution
Une **solution** d'un CSP est une instanciation complète S telle que toutes les contraintes de $\mathcal{C}$ sont satisfaites par S.

Exemple 1.1.
On considère le CSP suivant :
- $\mathcal{X} = \{x_1, x_2, x_3\}$,
- $\mathcal{D} = \{D_1, D_2, D_3\}$ avec $D_1 = D_2 = D_3 = \{1, 2, 3\}$,
- $\mathcal{C} = \{c_1, c_2\}$ avec $c_1 = [x_1 < x_2]$ et $c_2 = [x_2 = x_3]$.

Alors :

$var(c_1) = \{x_1, x_2\}$

$I_1 = \{(x_1, 1), (x_2, 2)\}$, $I_2 = \{(x_1, 1), (x_2, 3)\}$ et $I_3 = \{(x_1, 2), (x_2, 3)\}$ sont trois instanciations partielles satisfaisant c_1.

$var(c_2) = \{x_2, x_3\}$

$I'_1 = \{(x_2, 1), (x_3, 1)\}$, $I'_2 = \{(x_2, 2), (x_3, 2)\}$ et $I'_3 = \{(x_2, 3), (x_3, 3)\}$ sont trois instanciations partielles satisfaisant c_2.

Il existe trois solutions (instanciations complètes) satisfaisant c_1 et c_2 :
- $S_1 = \{(x_1, 1), (x_2, 2), (x_3, 2)\}$,
- $S_2 = \{(x_1, 1), (x_2, 3), (x_3, 3)\}$,
- $S_3 = \{(x_1, 2), (x_2, 3), (x_3, 3)\}$

1.2 Filtrage par Arc-Cohérence

Les CSP sont le plus souvent résolus par des méthodes de recherche arborescente où des sous-arbres sont élagués par filtrage. Le filtrage consiste à identifier certaines valeurs ne pouvant apparaître dans une solution et à les retirer des domaines correspondants. Les sous-arbres associés ne seront alors pas développés par l'algorithme de recherche. Ainsi, le filtrage permet d'obtenir un nouveau CSP équivalent au CSP de départ (ils possèdent exactement les mêmes solutions), mais dont la taille de l'arbre de recherche sera bien plus petite.

Enfin, les algorithmes de filtrage doivent être de complexité polynomiale. En effet, le temps gagné (en évitant de parcourir inutilement certains sous-arbres) doit être supérieur au temps nécessaire pour réaliser les filtrages successifs. Le filtrage le plus communément utilisé est le filtrage par Arc-Cohérence (`AC`). Sa complexité temporelle, dans le pire cas, est en $\mathcal{O}(e \times d^2)$, avec e nombre de contraintes et d taille du plus grand domaine.

Définition 1.4. Support d'une valeur pour une contrainte
Étant donné une contrainte binaire c telle que : $var(c) = \{x_1, x_2\}$. La valeur (x_j, v_j) est un **support** de la valeur (x_i, v_i) pour la contrainte c si et seulement si (v_i, v_j) satisfait c.

Définition 1.5. Viabilité d'une valeur
Une valeur est **viable** si et seulement si elle possède au moins un support pour chacune des contraintes portant sur elle.

Définition 1.6. Arc-Cohérence d'un CSP
Un CSP est arc-cohérent si et seulement si aucun domaine n'est vide et si toutes les valeurs des domaines sont viables.

Exemple 1.2.
Le filtrage par arc-cohérence du CSP de l'exemple 1.1 conduit aux domaines suivants :

— $D_1 = \{1, 2\}$,
— $D_2 = \{2, 3\}$,
— $D_3 = \{2, 3\}$.

Algorithme 1 : Fonction `Revise` (cas des contraintes binaires)

Données : x_i : variable, x_j : variable, D_i : domaine de x_i
Résultat : booléen

```
1 éliminé ← Faux;
2 pour tous les v ∈ D_i faire
3     si ∄ v_j ∈ D_j t.q. (v, v_j) satisfait c_i alors
4         D_i ← D_i\{v};
5         éliminé ← Vrai;
6 retourner éliminé ;
```

De nombreux algorithmes de filtrage ont été proposés pour rendre un CSP arc-cohérent. `AC-3` fut l'une des premières méthodes proposées [Mac77]. La fonction `Revise`(x_i, x_j, D_i) (cf. l'algorithme 1) retire du domaine D_i de la variable x_i les valeurs sans support pour la contrainte $c_{i,j}$. La complexité temporelle de `Revise` est en $\mathcal{O}(d^2)$ dans le pire cas.

`AC-3` (cf. l'algorithme 2) gère un ensemble Q de contraintes à traiter. Initialement, Q contient tous les couples $(c_{i,j}, x_i)$. Tant que Q n'est pas vide, un même traitement est effectué : le premier couple $(c_{i,j}, x_i)$ de Q est sélectionné ; puis, on tente de réduire le domaine D_i de la variable x_i à l'aide de la fonction `Revise`. Si D_i a été modifié par `Revise`, alors tous les couples $(c_{k,i}, x_k)$ avec $k \neq j$, non présents dans Q, sont ajoutés. Si le domaine d'une variable devient vide, alors `AC-3` retourne `Echec` signalant l'absence de solution. Lorsque Q devient vide, l'algorithme s'arrête et le CSP est arc-cohérent. `AC-3` maintient la cohérence d'arc en $\mathcal{O}(e \times d^3)$.

Plusieurs améliorations de `AC-3` ont été proposées :

— `AC-4` [MH86] a été la première méthode permettant d'établir l'AC en temps optimal dans le pire cas, i.e. en $\mathcal{O}(e \times d^2)$. `AC-4` calcule, tout d'abord, tous les supports de toutes les valeurs, puis effectue le filtrage proprement dit en $\mathcal{O}(e \times d^2)$.
— `AC-6` [BC93] a permis de diminuer la complexité en moyenne en gérant de manière paresseuse les supports. Au lieu de calculer l'ensemble de tous les supports pour toutes les valeurs, `AC-6` s'assure seulement qu'il existe au moins un support pour chaque valeur. Lorsqu'un tel support est retiré `AC-6` ne recherche plus un nouveau support à partir du début

Algorithme 2 : Fonction `AC-3` (cas des contraintes binaires)

Données : $\mathcal{X}$: ensemble de variables, $\mathcal{D}$: ensemble de domaines, $\mathcal{C}$: ensemble de contraintes
Résultat : {`Echec`, `Succes`}

```
1 Soit Q = {(c_{i,j}, x_i) t.q. c_{i,j} ∈ C, x_i ∈ X}
2 tant que Q ≠ ∅ faire
3     Extraire (c_{i,j}, x_i) de Q;
4     si Revise(x_i, x_j, D_i) alors
5         si D_i = ∅ alors
6             retourner Echec;
7         Q ← Q ∪ {(c_{k,i}, x_k) t.q. k ≠ j};
8 retourner Succes;
```

du domaine, mais à partir de celui venant d'être retiré.

— `AC-2001` [BR01] améliore la complexité en moyenne en stockant, dans une structure de données additionnelle, le dernier support obtenu et en testant, toujours en premier, la présence de celui-ci.

1.3 Résolution

Les méthodes de recherche arborescente reposent généralement sur le principe du retour arrière) (*Backtrack*) [GB65] associé à des méthodes de filtrage permettant d'élaguer l'arbre de recherche. De telles méthodes de recherche arborescente pour les CSP sont sûres (on n'obtient que des solutions) et complètes (on obtient toutes les solutions).

Dans cette section, nous rappelons le parcours fondé sur le Backtrack chronologique ainsi que le parcours avec Maintien de l'Arc-Cohérence (`MAC` [SF94, BR96]) qui réalise un filtrage par `AC` à chaque nœud de l'arbre.

1.3.1 Backtrack chronologique

Les variables du problème sont instanciées avec les valeurs de leur domaine, dans un ordre prédéfini, jusqu'à ce que l'un de ces choix viole une contrainte. La dernière instanciation effectuée est alors remise en cause. Une nouvelle valeur est essayée pour la dernière variable instanciée (variable courante). Si toutes les valeurs du domaine de cette variable ont été visitées, on doit procéder à un retour en arrière. Pour cela, une nouvelle valeur est choisie pour la variable précédant immédiatement la variable courante. Ce processus est répété jusqu'à obtenir une solution, i.e. une instanciation de toutes les variables ne violant aucune contrainte. Si on a parcouru tout l'arbre de recherche sans en trouver, alors le problème ne possède aucune solution.

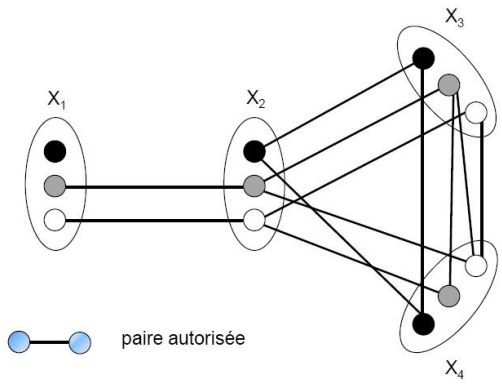

FIGURE 1.1 – Graphe de cohérence

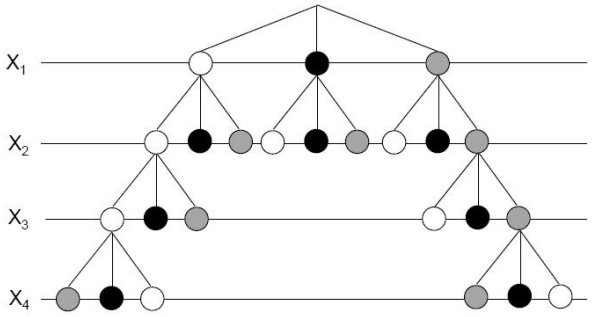

FIGURE 1.2 – Arbre de recherche parcouru par backtrack

Exemple 1.3.
Soit le CSP décrit par son graphe de cohérence [1] à la figure 1.1. Ce CSP n'ayant qu'une unique solution (gris, gris, gris, blanc), l'ordre de parcours des valeurs a été choisi de manière à explorer l'intégralité de l'arbre de recherche pour trouver cette solution. L'arbre de recherche par Backtrack chronologique est décrit à la figure 1.2.

1.3.2 Maintien d'arc-cohérence

Supposons qu'en cherchant une solution, le Backtrack chronologique donne à une variable x_i une valeur v qui exclut toutes les valeurs possibles pour une variable x_j. Cet algorithme ne s'en rendra compte que lorsque x_j sera considérée. De plus, avec un Backtrack chronologique, avant que x_i ne soit reconsidérée, il se peut qu'une grande partie de l'espace de recherche soit explorée en vain. Ceci

1. Le graphe de cohérence est un graphe dont les sommets sont les valeurs des domaines, et dont les arcs reliant les sommets représentent les supports.

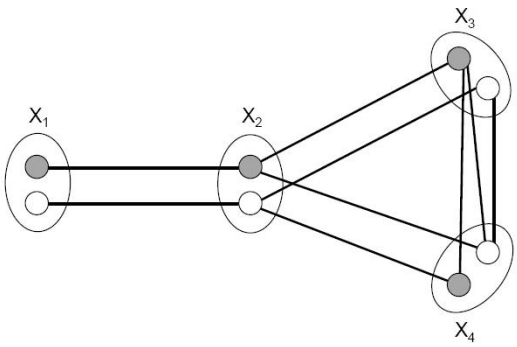

FIGURE 1.3 – Graphe de cohérence après filtrage par `AC`

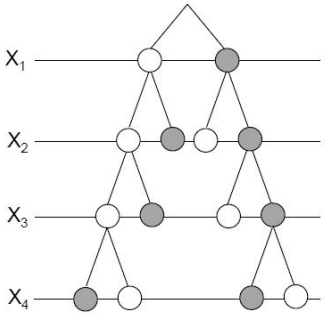

FIGURE 1.4 – Arbre de recherche parcouru par `MAC`

pourrait être évité en prenant en compte le fait que (x_i, v) ne pourra jamais faire partie d'une solution, puisqu'il n'y a aucune valeur de x_j qui soit compatible avec (x_i, v). Utiliser le filtrage permet d'éviter la situation décrite ci-dessus. Une valeur v n'est acceptée qu'après avoir vérifié les domaines des variables futures. S'il existe un domaine qui ne contient que des valeurs incompatibles avec v, alors v est éliminée. Par ailleurs, le filtrage peut être utilisé après chaque instanciation (x_i, v). En effet, les valeurs des variables futures qui ne sont pas compatibles avec (x_i, v) peuvent être éliminées sans perte de solutions.

L'algorithme de Maintien d'Arc Cohérence (`MAC`) procède au au filtrage par `AC` après chaque instanciation de variable [SF94, BR96]. L'algorithme 3 montre comment un CSP peut être résolu en utilisant une recherche en profondeur d'abord. D et C désignent respectivement l'ensemble des domaines courants et l'ensemble courant de contraintes. `Filter`(D, C) effectue le filtrage des domaines de D selon les contraintes de C (ligne 1). A chaque nœud de l'arbre de recherche, l'algorithme fait le branching en instanciant une variable par une valeur de son domaine courant (lignes 6 et 7). Il revient en arrière en cas de violation d'une des contraintes, i.e. lorsqu'au moins un domaine est vide (ligne 2). La recherche

Algorithme 3 : Procédure $Depth\text{-}First(D, C)$

Données : D : ensemble de domaines, C : ensemble de contraintes

```
1 D ← Filter(D, C);
2 si il existe x_i ∈ 𝒳 tel que D_i = ∅ alors
3     retourner Échec;
4 si il existe x_i ∈ 𝒳 tel que |D_i| > 1 alors
5     Sélectionner x_i ∈ 𝒳 tel que |D_i| > 1;
6     pour tous les v ∈ D_i faire
7         Depth-First(D ∪ {(x_i, v)}, C);
8 sinon
9     Output(D, C);
```

peut être améliorée en choisissant avec soin [2] la variable qui sera affectée (ligne 5). On obtient une solution (ligne 9) lorsque chaque domaine D_i est réduit à un singleton et toutes les contraintes sont satisfaites.

1.4 Contraintes réifiées

Une contrainte réifiée associe une variable x de domaine $\{0, 1\}$ à une contrainte c de manière à refléter la satisfaction de la contrainte (valeur 1), ou sa non-satisfaction (valeur 0). Par exemple, x vaut 1 ssi la contrainte c est satisfaite :

$$(x = 1) \iff c \quad \text{avec } D_{x_i} = \{0, 1\} \text{ et } x_i \notin var(c)$$

De telles contraintes sont très utiles pour exprimer des formules booléennes sur des contraintes et pour exprimer qu'un certain nombre de contraintes doivent être satisfaites.

Le filtrage des contraintes réifiées s'effectue comme suit :

1. Pour la première implication :
   — si x vaut 1 alors la contrainte c doit être satisfaite,
   — si x vaut 0 alors la contrainte $\neg c$ doit être satisfaite,
2. Pour la deuxième implication :
   — si la contrainte c est satisfaite alors x devra valoir 1.
   — si la contrainte c n'est pas satisfaite alors alors x devra valoir 0.

Les contraintes réifiées seront utilisées au chapitre suivant pour modéliser les problèmes d'extraction de motifs ensemblistes.

2. Par exemple, en utilisant l'heuristique *dom/deg* qui sélectionne la variable de plus petit ratio entre la taille de son domaine courant et le nombre de contraintes où elle figure.

Bibliographie

[BC93] Christian Bessière and Marie-Odile Cordier. Arc-consistency and arc-consistency again. In Richard Fikes and Wendy G. Lehnert, editors, *AAAI*, pages 108–113. AAAI Press / The MIT Press, 1993.

[BR96] Christian Bessière and Jean-Charles Régin. MAC and combined heuristics : Two reasons to forsake FC (and CBJ ?) on hard problems. In Eugene C. Freuder, editor, *CP*, volume 1118 of *Lecture Notes in Computer Science*, pages 61–75. Springer, 1996.

[BR01] Christian Bessière and Jean-Charles Régin. Refining the basic constraint propagation algorithm. In Bernhard Nebel, editor, *IJCAI*, pages 309–315. Morgan Kaufmann, 2001.

[GB65] Solomon W. Golomb and Leonard D. Baumert. Backtrack programming. *J. ACM*, 12(4) :516–524, 1965.

[Mac77] Alan K. Mackworth. Consistency in networks of relations. *Artif. Intell.*, 8(1) :99–118, 1977.

[MH86] Roger Mohr and Thomas C. Henderson. Arc and path consistency revisited. *Artif. Intell.*, 28(2) :225–233, 1986.

[SF94] Daniel Sabin and Eugene C. Freuder. Contradicting conventional wisdom in constraint satisfaction. In Alan Borning, editor, *PPCP*, volume 874 of *Lecture Notes in Computer Science*, pages 10–20. Springer, 1994.