PROGRAMMATION PAR CONTRAINTES

Xavier Olive

basé sur les supports de Cédric Pralet

https://xolearn.github.io/constraints

OBJECTIFS DU COURS

La programmation par contraintes définit un formalisme pour définir des problèmes d'optimisation sous contraintes.

Objectifs du cours:

- comprendre et manipuler le formalisme (cours/BE)
- comprendre les principales méthodes de résolution (cours)
- modéliser un problème complexe à l'aide de ce formalisme (projet noté)

- Programmation linéaire (simplexe, points intérieurs)
- Programmation linéaire en nombres entiers (recherche arborescente)
- Programmation non linéaire (méthodes de gradient, estimation de densité)

La programmation par contraintes permet de manipuler des **contraintes non linéaires sur des variables discrètes**.

Les méthodes de résolution combinent **recherche arborescente** et **propagation de contraintes**.

	Abréviations
PPC	Programmation par contraintes
CP	*Constraint Programming*
CSP	*Constraint Satisfaction Problem*
COP	*Constraint Optimisation Problem*

DÉFINITIONS

Un problème CSP (V, D, C) est composé de:

- $V = (v_1, v_2, \dots v_n)$, les variables,
- $D = (d_1, d_2, \dots d_n)$, les domaines finis pour chacune des variables de V;
- $C = (c_1, c_2, \dots c_m)$, une séquence de contraintes, chacune définie par un couple (s_i, r_i):
  - s_i est une séquence de variables;
  - r_i est une relation définie par un sous-ensemble du produit cartésien $d_{i_1} \times \dots d_{i_{n_i}}$ de valeurs autorisées.

Définition: scope et arité

Soit $c = (s, r)$, s est aussi appelé scope de la contrainte. La taille du scope est appelée arité de la contrainte.

Définition: intension/extension

On peut définir les contraintes en extension:

$$C : \left(\left(x, y\right), \left\{\left(0, 1\right), \left(0, 2\right), \left(1, 0\right), \left(1, 2\right), \left(2, 0\right), \left(2, 1\right)\right\}\right)$$

ou en intension:

$$C : x \neq y \text{ avec } x, y \in \{0, 1, 2\}$$

Définition: instanciation

Étant donné un CSP (V, D, C), on appelle instanciation $\mathcal{A}$ de $Y = \left\{v_{y_1}, \dots v_{y_m}\right\} \subset V$, une application qui associe à chaque variable v_{y_i} une valeur $\mathcal{A}\left(v_{y_i}\right) \in d_{y_i}$.

Définition: satisfaction de contrainte

Une instanciation $\mathcal{A}$ de Y satisfait la contrainte $c_i = (s_i, r_i)$ de C:

$$\mathcal{A} \vDash c_i \Leftrightarrow s_i \subset Y \land \mathcal{A}\left(s_i\right) \in r_i$$

À l'opposé, on parle de violation de contrainte.

INSTANCIATION COHÉRENTE

Une instanciation $\mathcal{A}$ de $Y \subset X$ est cohérente ssi pour toute contrainte $c_i = (s_i, r_i) \in C$ telle que $s_i \subset Y, \mathcal{A} \vDash c_i$.

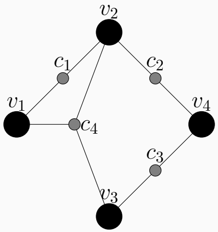
Une instanciation est cohérente si elle ne viole aucune contrainte.

- Une solution est une instanciation cohérente de X.
- Un CSP est cohérent s'il a au moins une solution.

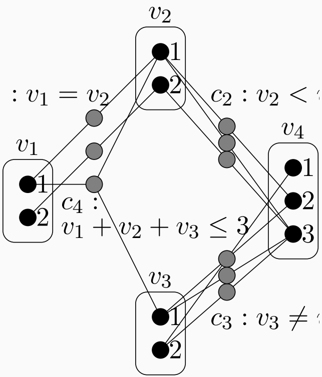
Une instanciation $\mathcal{A}$ de $Y \subset X$ est globalement cohérente ssi il existe une solution $\mathcal{S}$ telle que $\mathcal{A} \subset \mathcal{S}$.

Si une instanciation n'est pas globalement cohérente, il n'est pas possible de l'étendre en une solution.

Représentation sous forme d'un graphe de contraintes



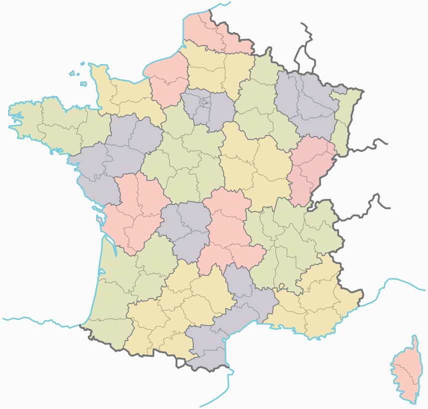
- sommets: les variables et les contraintes du CSP
- arité d'une contrainte: degré des sommets *contrainte*
- degré d'une variable: degré des sommets *variable*

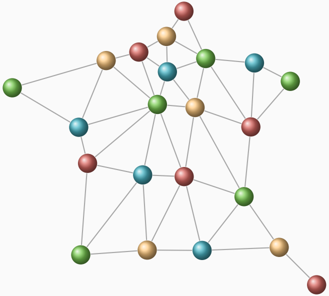


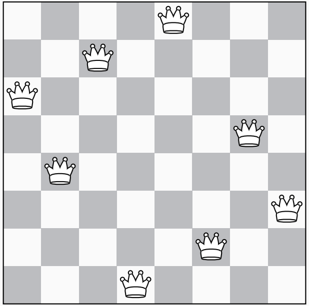
- Représentation explicite des domaines et combinaisons de valeurs autorisées

EXEMPLES

Comment colorer la carte de sorte que deux régions voisines soient de couleurs différentes, en utilisant au plus k couleurs?







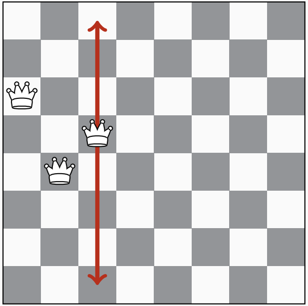
Comment placer n reines sur un échiquier $n \times n$ de sorte qu'aucune reine n'en attaque une autre?

7	5	8		3				
	4			8		3	7	
		3			2		8	
					4	1		3
4			3		5			8
3		7	8					
	6		1					
	3	5		9			4	
				5		8	9	1

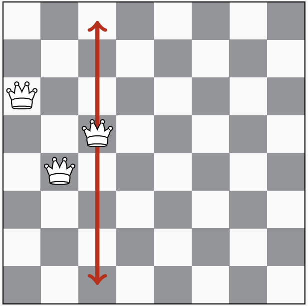
Placer des chiffres qui doivent être tous différents sur chaque ligne, chaque colonne et chaque sous-carré.

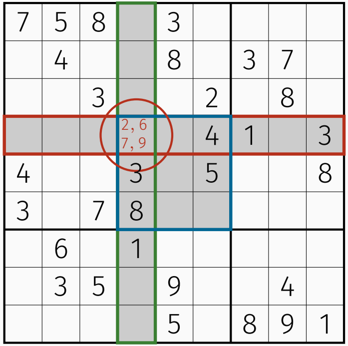
RÉSOLUTION D’UN CSP

- Même avec peu de variables et de contraintes, l'espace de recherche est tel qu'il conduit à phénomène d'explosion combinatoire (complexité $O(m \cdot d^n)$ avec n variables, d la taille max des domaines et m contraintes.)
- Problème NP-complet: pas d'algorithme complet connu pour le problème CSP dont la complexité serait polynomiale.



7	5	8		3				
	4			8		3	7	
		3			2		8	
			2, 6 7, 9		4	1		3
4			3		5			8
3		7	8					
	6		1					
	3	5		9			4	
				5		8	9	1

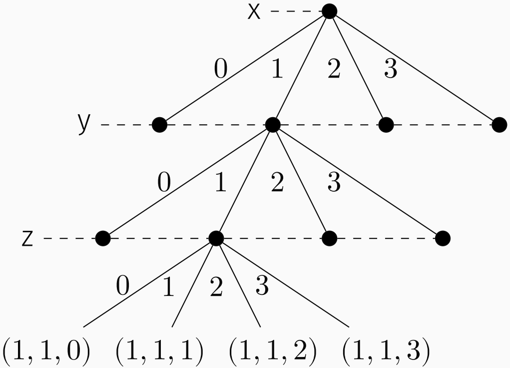


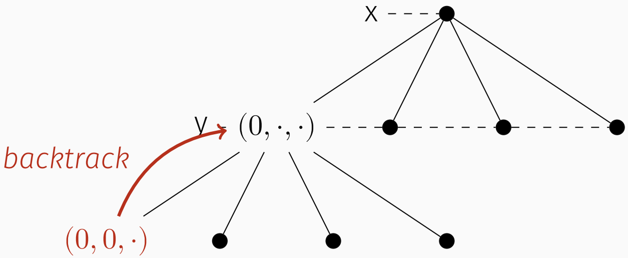


- Résolution par recherche arborescente et propagation de contraintes.

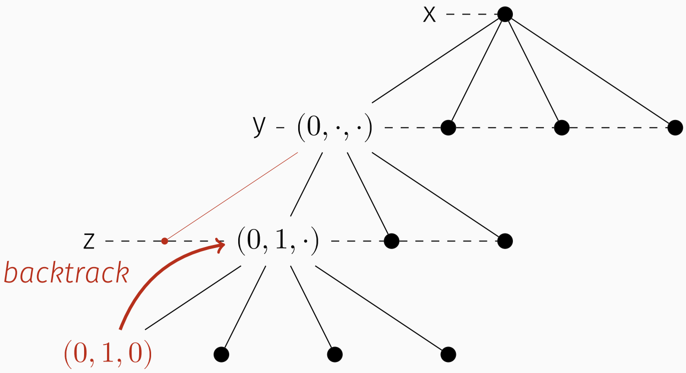
- On parcourt l'espace de recherche des solutions en affectant des valeurs à des variables.
  Une affectation constitue une *hypothèse*.
- L'ensemble des hypothèses faites à une étape donnée constitue une instanciation partielle, dont on vérifie la cohérence.
- On **parcourt en profondeur un arbre de recherche (DFS)**:
  - un nœud n correspond à l'affectation d'une variable x;
  - une arête issue de n correspond à une valeur a affectée à x;
  - les feuilles sont des instanciations complètes.
- Si une contrainte portant sur les variables déjà affectées est violée (*backward checking*), on remonte dans l'arbre.

EXEMPLE

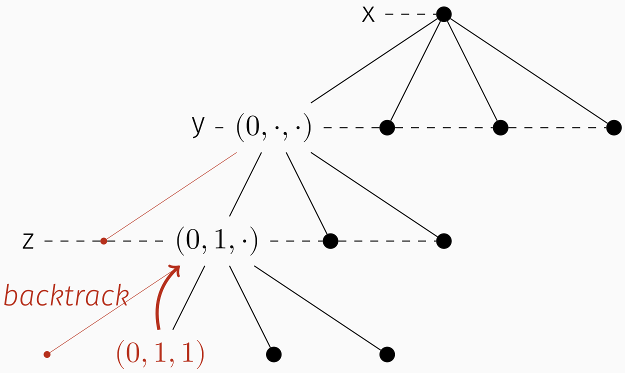


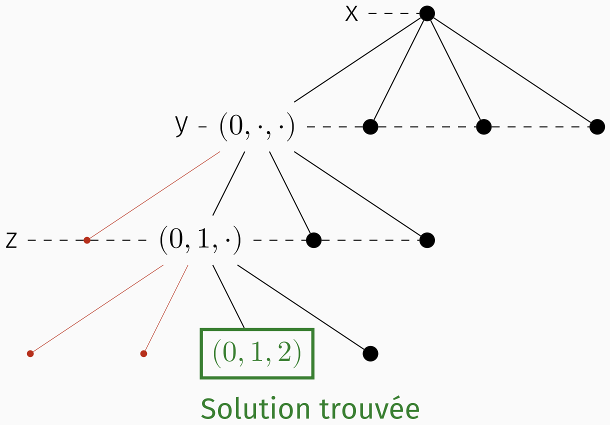


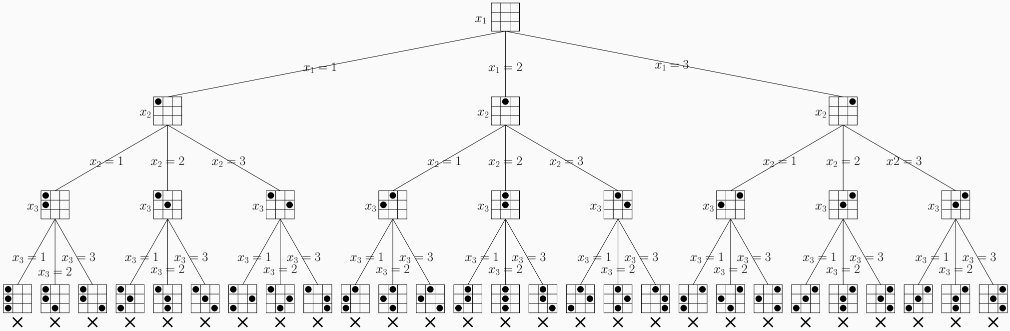
- L'assignation partielle $(0, 0, \cdot)$ viole la contrainte $x \neq y$: on interrompt le parcours en profondeur. On parle alors de backtracking.


x
y
$(0, \cdot, \cdot)$
z
$(0, 1, \cdot)$
backtrack
$(0, 1, 0)$

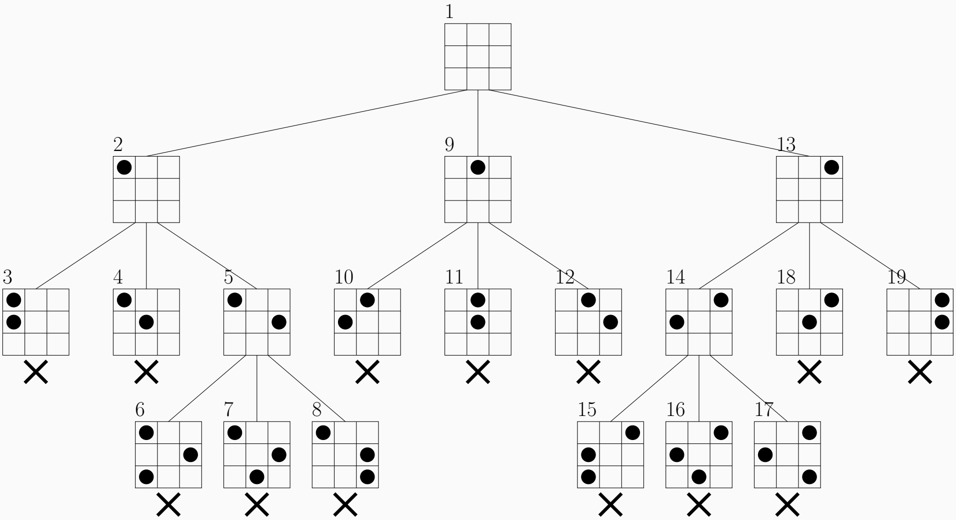
EXEMPLE




x
y
$(0, \cdot, \cdot)$
z
$(0, 1, \cdot)$
$(0, 1, 2)$
Solution trouvée



- 3 reines: 27 feuilles; 8 reines: plus de 16 millions de feuilles
- n reines: n^n feuilles


1
2
9
13
3
4
5
10
11
12
14
18
19
6
7
8
15
16
17

PROPAGATION DE CONTRAINTES

Idée poursuivie: effectuer des raisonnements plus poussés à chaque nœud de l'arbre de recherche pour essayer de **détecter plus tôt les incohérences**

En particulier, détection d'incohérence par raisonnement sur des contraintes **avant que toutes les variables** de ces contraintes ne soient instanciées.

- pour $c : x = y$ et avec $d_x = \{0, 1\}$, $d_y = \{2, 3\}$, incohérence détectable immédiatement
- pour $C = \{c_1, c_2, c_3\}$ avec $c_1 : x < y$, $c_2 : y < z$ et $c_3 : z < x$, incohérence détectable également (incohérence due à des interactions entre contraintes)

On parle alors de **propagation de contraintes**

Les techniques de cohérence locale ont pour objectif de faire des déductions pour **simplifier un problème donné**.

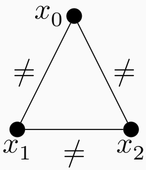
On effectue des raisonnements locaux afin de déduire qu'une assignation de valeur à une variable ne participe à aucune solution, et qu'il est donc possible de **supprimer cette valeur du domaine**.

L'arc-cohérence est la plus simple et la plus utilisée des cohérences locales: elle correspond à la **cohérence locale sur toutes les contraintes binaires**.

Un CSP (V, D, C) est arc-cohérent ssi pour toute variable $x \in V$, et pour toute contrainte {binaire} $c = (\{x, y\}, R) \in C$, on a:

$$\forall a \in d(x)\ \exists b \in d(y),\ (x, a), (y, b) \in R$$

Un problème qui n'est pas arc-cohérent ne sera pas cohérent.
La réciproque n'est pas vraie.



1. $x_0, x_1, x_2 \in \{0, 1\}$
2. En assignant une valeur à x_0, on peut toujours assigner des valeurs à x_1, x_2 sans violer les contraintes qui impliquent x_0.

Fonction de base:
révision du domaine d_x d'une variable x en raisonnant sur une contrainte c_{xy}

Algorithm 1 `revise`(x, y)

$change \leftarrow false$
for $a \in d_x$ **do**
  if $\nexists b \in d_y,\ \{(x,a),(y,b)\}$ *satisfies* c_{xy} **then**
    delete a from y
    $change \leftarrow true$
return $change$

Idée générale: maintien d'une liste de révisions à effectuer

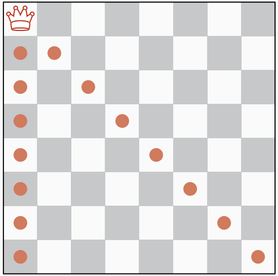
Algorithm 2 AC3(V, C)

$Q \leftarrow \{(x, y) \mid c_{xy} \in C\}$ # liste des couples (x, y) à réviser
while $Q \neq \emptyset$ **do**
  $(x, y) \leftarrow$ remove one element from Q
  $change \leftarrow$ `revise`(x, y)
  if $change$ **then**
    if $d_x = \emptyset$ **then return** $false$ # preuve d'incohérence
    $Q \leftarrow Q \cup \{(z, x) \mid c_{zx} \in C \wedge z \neq y\}$ # révisions requises sur les voisins de x
return $true$

On utilise généralement les procédures AC* lors d'une recherche arborescente, pour s'assurer qu'après toute assignation, le sous-problème induit reste arc-cohérent.

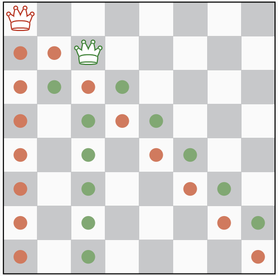
On utilise souvent une version *dégradée* de l'arc-cohérence. À l'étape x_i, on peut vérifier l'arc-cohérence pour tout couple:

- (x_k, x_i) tel que $i < k \leq n$ (*forward-checking*)
- (x_j, x_k) tel que $i \leq j < k \leq n$ (*partial look-ahead*)
- (x_j, x_k) tel que $i \leq j \neq k \leq n$ (*full look-ahead*)



Forward-checking

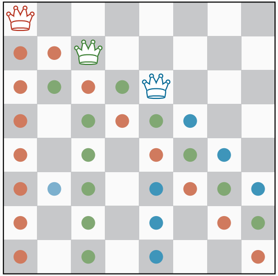
À partir d'une variable x_i donnée, on vérifie l'arc-cohérence pour tous les couples (x_k, x_i) tels que $i < k \leq n$.



Forward-checking

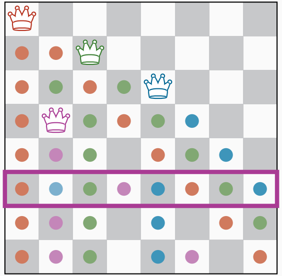
À partir d'une variable x_i donnée, on vérifie l'arc-cohérence pour tous les couples (x_k, x_i) tels que $i < k \leq n$.

EXEMPLE SUR LE PROBLÈME DES n REINES



Forward-checking

À partir d'une variable x_i donnée, on vérifie l'arc-cohérence pour tous les couples (x_k, x_i) tels que $i < k \leq n$.



Forward-checking

À partir d'une variable x_i donnée, on vérifie l'arc-cohérence pour tous les couples (x_k, x_i) tels que $i < k \leq n$.

- Plus de valeur possible pour x_6: backtrack

Points forts:

- possibilité de l'assurer à moindre coût, en $O(m \cdot d^2)$ ($O(m \cdot d^3)$ pour AC-3);
- possibilité de l'assurer avant la recherche arborescente ou pendant;

Points faibles:

- insuffisante pour garantir l'existence d'une solution (possibilité de ne pas détecter certaines incohérences)
- applicable uniquement aux contraintes binaires

CONTRAINTES GLOBALES

- Les contraintes globales sont des contraintes particulières avec des mécanismes de propagation optimisés;
- Ces contraintes apportent une sémantique riche;
  leur efficacité rendent la PPC compétitive sur des problèmes difficiles;
- Catalogue de contraintes globales: http://sofdem.github.io/gccat/

`alldifferent`$(x_1, \ldots, x_k)$ (porte sur k variables)

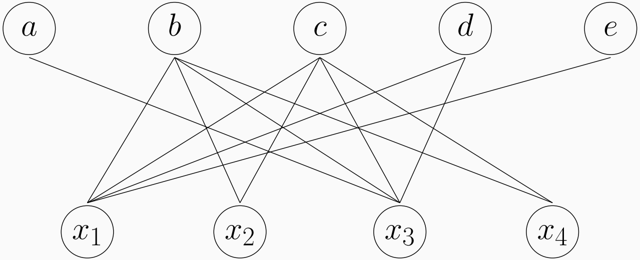
satisfaite ssi: $\forall i \neq j \in [1..k]$, $x_i \neq x_j$

Exemple:

$$x_1 \in \{b, c, d, e\}$$
$$x_2 \in \{b, c\}$$
$$x_3 \in \{a, b, c, d\}$$
$$x_4 \in \{b, c\}$$
$$\texttt{alldifferent}(x_1, x_2, x_3, x_4)$$

Établissement facile de l'arc-cohérence généralisée grâce à des techniques d'analyse de graphes

Pour propager la contrainte, on part d'un graphe bipartite:



L'affectation courante est cohérente ssi il existe un **k-matching** dans le graphe bi-partite associé (pour k variables)

L'algorithme de Hopcroft et Karp, teste l'existence d'un k-matching dans un graphe bipartite en $O(s \cdot \sqrt{k})$, avec s somme des tailles des domaines de valeurs

On peut alors supprimer les valeurs non arc-cohérentes en temps polynomial $O(s)$; on peut notamment supprimer rapidement les choix $x_1 = b$, $x_1 = c$, $x_3 = b$, $x_3 = c)\}$

ALLDIFFERENT

7	5	8		3			1, 2 6	
	4			8		3	7	
		3			2		8	
					4	1		3
4			3		5			8
3		7	8					
	6		1					
	3	5		9			4	
				5		8	9	1

par arc-cohérence

7	5	8		3			1	
	4			8		3	7	
		3			2		8	
					4	1		3
4			3		5			8
3		7	8					
	6		1					
	3	5		9			4	
				5		8	9	1

par k-matching

Exemple d'une contrainte `noOverlap`(T) de **non-chevauchement entre tâches** à réaliser sur une ressource disjonctive (non partageable)

Entrées de la contrainte: un ensemble de tâches T non interruptibles, avec $\forall t \in T$:

- une durée de réalisation p_t (*processing time*)
- une date de début au plus tôt est_t (*earliest start time*)
- une date de fin au plus tard let_t (*latest end time*)

Variables de décision manipulées par la contrainte: pour chaque tâche $t \in T$,

- variable *date de début* sta_t
- variable *date de fin* end_t $(= sta_t + p_t)$

Domaines de valeurs:

$d_{sta_t} = [est_t, let_t - p_t]$ et $d_{end_t} = [est_t + p_t, let_t]$

Contrainte `noOverlap`(T) satisfaite si et seulement si:

$$\forall t, t' \in T,\ t \neq t',\ (sta(t) \geq end(t')) \vee (sta(t') \geq end(t))$$

(contrainte portant sur $k = 2n$ variables avec n le nombre de tâches)

Principe: recherche de contraintes de précédence induites entre tâches pour filtrer les domaines de valeurs des variables sta_t / end_t

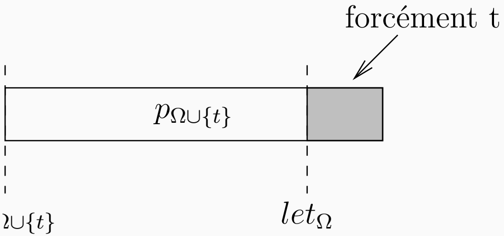
Notations: pour $\Omega \subseteq T$,

- $est_\Omega = min_{t \in \Omega} est_t$ (date de début au plus tôt d'une tâche dans Ω)
- $let_\Omega = max_{t \in \Omega} let_t$ (date de fin au plus tard d'une tâche dans Ω)
- $p_\Omega = \sum_{t \in \Omega} p_t$ (somme des durées des tâches dans Ω)

Règle de propagation des dates au plus tôt: pour toute tâche $t \in T$ et tout ensemble de tâches $\Omega \subseteq T \quad \{t\}$,

Si $est_{\Omega \cup \{t\}} + p_{\Omega \cup \{t\}} > let_{\Omega}$, alors $t \gg \Omega$ (tâche t située après les tâches de Ω)

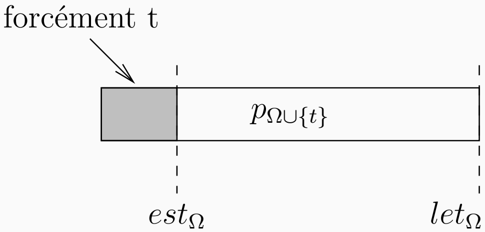
Dans ce cas, filtrage $est_t \leftarrow max(est_t, est_\Omega + p_\Omega)$

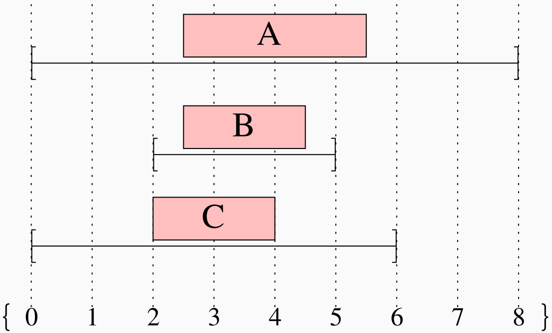


Règle de propagation des dates au plus tard: pour toute tâche $t \in T$ et tout ensemble de tâches $\Omega \subseteq T \quad \{t\}$,

Si $let_{\Omega \cup \{t\}} - p_{\Omega \cup \{t\}} < est_{\Omega}$, alors $t \ll \Omega$ (tâche t située avant les tâches de Ω)

Dans ce cas, filtrage $let_t \leftarrow min(let_t, let_{\Omega} - p_{\Omega})$

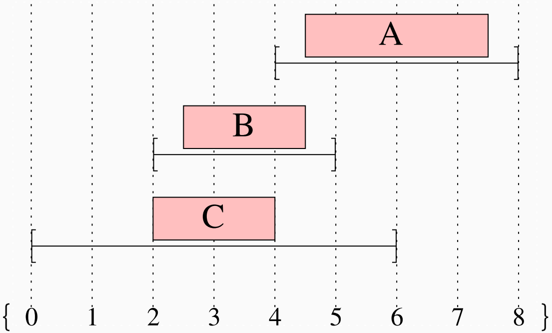




Pour $t = A$ et $\Omega = \{B, C\}$:

$est_{A,B,C}(0) + p_{A,B,C}(7) > let_{B,C}(6)$

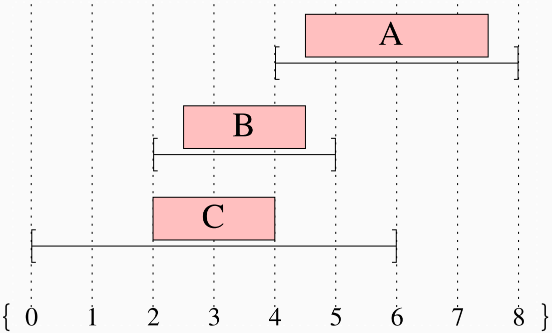
Déduction: $A \gg \{B, C\}$, donc $est_A \leftarrow max(est_A(0), est_{B,C}(0) + p_{B,C}(4))$



Pour $t = A$ et $\Omega = \{B, C\}$:

$est_{A,B,C}(0) + p_{A,B,C}(7) > let_{B,C}(6)$

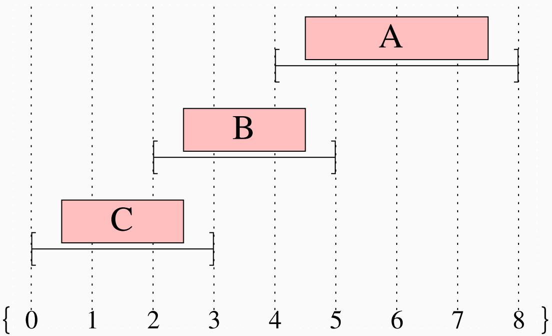
Déduction: $A \gg \{B, C\}$, donc $est_A \leftarrow max(est_A(0), est_{B,C}(0) + p_{B,C}(4))$



Pour $t = C$ et $\Omega = \{A, B\}$:

$$let_{A,B,C}(8) - p_{A,B,C}(7) < est_{A,B}(2)$$

Déduction: $C \ll \{A, B\}$, donc $let_C \leftarrow min(let_C(6), eet_{A,B}(8) - p_{A,B}(5))$



Pour $t = C$ et $\Omega = \{A, B\}$:

$let_{A,B,C}(8) - p_{A,B,C}(7) < est_{A,B}(2)$

Déduction: $C \ll \{A, B\}$, donc $let_C \leftarrow min(let_C(6), eet_{A,B}(8) - p_{A,B}(5))$

In fine, on prouve que $C \ll A \ll B$ par propagation

Remarque: des raisonnements disjoints sur les contraintes $(sta(t) \geq end(t')) \vee (sta(t') \geq end(t))$ pour $t \neq t' \in \{A, B, C\}$ n'auraient rien donné en termes de propagation

D'où l'intérêt d'avoir des techniques de propagation de contraintes spécifiques raisonnant sur des connaissances globales présentant une structure particulière

BONNES PRATIQUES

Taille de l'arbre de recherche exploré fonction de l'ordre choisi pour affecter les variables et de l'ordre dans lequel les valeurs sont choisies

Exemple d'**heuristiques de choix de variable** ($\equiv$ guides):

- choix d'une variable de plus petit domaine courant (**Min-Domain**)
- choix d'une variable impliquée dans le plus de contraintes (**Max-Degree**)
- choix d'une variable minimisant le ratio taille du domaine par degré (**Min-Domain / Max-Degree**)

Principe général: principe **fail-first** (pour détecter des échecs le plus tôt possible)

En général, bénéfique d'utiliser des **heuristiques dynamiques**:

Exemples:

- plus petit domaine courant au lieu de plus petit domaine initial
- variables liées avec le plus de contraintes non instanciées étant donné l'affectation courante
- maintien d'un poids associé à chaque contrainte en fonction des échecs rencontrés et choix de variable en fonction de ces poids (**Weighted-Degree**)

Heuristique de choix de valeur: ordre dans lequel les valeurs sont sélectionnées

Principe first-success: sélection de la valeur la moins contraignante d'abord, pour trouver des solutions le plus tôt possible

Diversité des types de branchement:

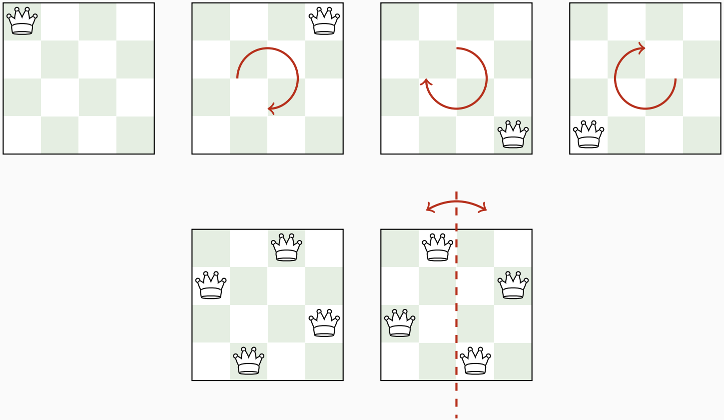
- branchement binaire $x = a$ et $x \neq a$
- branchement dichotomique $x \leq a$ et $x > a$
- branchement énumératif $x = 1$, $x = 2$, ...$x = m_x$

Remarque: très dépendant de l'application (possibilité de définir des heuristiques métier)

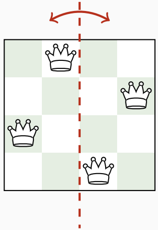
Souvent plusieurs modèles candidats et choix des variables très important (détermine l'espace de recherche)

Possibilité d'ajouter des contraintes en plus du modèle de base (idem PLNE):

- contraintes redondantes pour accélérer la recherche (contraintes induites qui seraient difficiles à trouver pour les outils et qui se propagent bien)
- contraintes pour casser les symétries (élimination de solutions équivalentes)
- contraintes supprimant des solutions sous-optimales
- contraintes supprimant des solutions optimales mais pas toutes

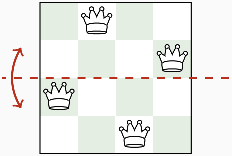


Une symétrie σ sur un CSP est un automorphisme sur l'ensemble des assignations qui laisse l'ensemble des contraintes globalement inchangé.



σ est une permutation de variables:

- $x_1 \rightleftharpoons x_4$
- $x_2 \rightleftharpoons x_3$



σ est une permutation de valeurs:

- $1 \rightleftharpoons 4$
- $2 \rightleftharpoons 3$

On peut exploiter les symétries d'un problème pour réduire le domaine de recherche:

- reformulation du problème;
- ajout statique de contraintes;
- ajout dynamique de contraintes;
- détection de dominance

Tout l'enjeu consiste à déterminer si détecter les symétries reste moins coûteux que ce que leur exploitation peut rapporter.

RECHERCHE INCOMPLÈTE

RECHERCHE INCOMPLÈTE

On accepte de ne pas trouver l'optimum global, mais plutôt de **trouver des bonnes solutions rapidement**, en privilégiant de la recherche dans des voisinages *prometteurs*.

Exploration plus libre et plus diversifiée.

Deux techniques présentées ici:

- min-conflicts;
- large neighbourhood search (LNS)

On accepte de relâcher certaines contraintes au début. On travaille sur une amélioration itérative basée sur la minimisation du nombre de **contraintes non satisfaites**, avec arrêt lorsque ce nombre vaut 0.

- choix d'une **affectation initiale** A_0 quelconque, puis
- choix aléatoire d'une variable x parmi les variables qui interviennent dans au moins une contrainte violée,
- choix d'une valeur a dans le domaine de x de manière à **minimiser le nombre de contraintes non satisfaites** après réaffectation de la variable x,
- nouvelle affectation A_{i+1} obtenue à partir de A_i en donnant la valeur a à x.

Pour **diversifier la recherche**, possibilité de faire des restarts à partir d'une nouvelle affectation initiale A_0

MIN-CONFLICTS

- Arrêt de l'algorithme quand une solution est trouvée ou quand un critère est atteint (p.ex. timeout)
- Possibilité de boucles dans la recherche, de rester bloqué dans des minima locaux
- Pas de garantie de trouver l'optimum
- Incapacité à trouver l'incohérence d'un problème

- choix d'une affectation initiale A_0 quelconque, puis
- choix de k variables $x_1, \ldots, x_k$ parmi les n variables du problème
- recherche complète de la meilleure réinstanciation de $x_1, \ldots, x_k$ étant donné les $n - k$ autres variables fixées à leur valeur courante (voisinage large)
- nouvelle affectation A_{i+1} obtenue à partir de A_i en donnant les meilleures valeurs trouvées à $x_1, \ldots, x_k$.

Avantages:

- complexité limitée de chaque recherche dans un voisinage large (complexité pire cas exponentielle en k et non en n)
- par rapport à min-conflicts, plus de chance de sortir des optima locaux (min-conflict $\equiv$ LNS avec $k = 1$)
- possibilité de faire varier k pendant la recherche
- utilisation de la puissance des méthodes complètes sur des instances de taille "raisonnable"

Paramètre à régler: méthode de choix des k variables à réinstancier à chaque étape

OUTILS

Plusieurs bibliothèques d'optimisation sous contraintes proposent:

- une API ou un langage de modélisation;
- des algorithmes de recherche et de propagation prédéfinis;
- des éléments pour paramétrer ces algorithmes;
- des éléments pour définir de nouvelles contraintes et de nouveaux types de branchements.

Parmi les plus célèbres: IBM ILOG CP Optimizer, Gecode, Choco, OR-Tools, etc.

Nous utiliserons dans les BE une interface Python pour un solver minimaliste libre et gratuit écrit en OCaml.