Chapitre 10

Programmation par contraintes

Louis-Martin Rousseau et Gilles Pesant

La programmation par contraintes (PPC) est, au sens large, l'étude des systèmes de calcul basés sur les contraintes. Ses origines mixtes se retrouvent en interfaces personne-machine dès les années soixante, en intelligence artificielle dans les années soixante-dix et en langages de programmation dans les années quatre-vingt. Son application sérieuse et répandue au domaine de la recherche opérationnelle est plus récente ; elle date des années quatre-vingt-dix. Dans ce chapitre, nous traitons une partie seulement de la programmation par contraintes, celle qui s'intéresse à la résolution de problèmes combinatoires. Nous présentons les concepts centraux regroupés selon les trois thèmes suivants : modélisation du problème (sect. 10.2), propagation des contraintes (sect. 10.3) et recherche de solutions (sect. 10.4). Le lecteur souhaitant une introduction plus générale à la PPC pourra consulter quelques ouvrages (Apt, [1] ; Dechter, [2] ; Marriot et Stuckey [3]).

10.1 PRINCIPE GÉNÉRAL

L'approche de la programmation par contraintes pour résoudre des problèmes combinatoires consiste à les modéliser par un ensemble de variables prenant leur valeur dans un ensemble fini et liées par un ensemble de contraintes mathématiques ou symboliques (sect. 10.2). L'efficacité de ce paradigme repose sur de

puissants algorithmes de propagation de contraintes qui éliminent du domaine des variables les valeurs qui engendrent des solutions irréalisables (sect. 10.3). Si la propagation de contraintes ne suffit pas à elle seule à établir une solution réalisable, une recherche arborescente est entreprise afin de réduire davantage le domaine des variables définissant le problème (sect. 10.4). À chaque nœud de l'arbre de recherche une variable est d'abord choisie selon une certaine politique (statique ou dynamique) puis fixée à une des valeurs possibles de son domaine. On a une solution lorsque le domaine de chaque variable ne contient qu'une seule valeur et une impasse lorsque le domaine d'une variable devient vide. Il est généralement possible de guider la recherche de solutions réalisables en incorporant de l'information sur la nature du problème dans les politiques de sélection de variables et de valeurs.

10.2 MODÉLISATION

10.2.1 Variables, domaines, contraintes

La résolution de problèmes combinatoires en PPC passe par une modélisation basée sur des *variables à domaine fini*. À chaque variable x est associé un ensemble fini de valeurs D_x, appelé son *domaine*, au sein duquel elle prend nécessairement sa valeur. Ce domaine est donné tantôt en extension :

$$x \in \{1, 2, 4, 6, 7, 11\}$$

tantôt en compréhension :

$$1 \leq x \leq 5$$

Un aspect très important de ces domaines est qu'ils ne sont pas simplement statiques, mais plutôt mis à jour au fur et à mesure que des décisions sont prises. Par exemple, $D_x = \{1, 2, 4, 6, 7, 11\}$ deviendra $D_x = \{1, 2, 4\}$ s'il appert que $x \leq 5$ dans le sous-problème considéré. Chaque domaine est donc maintenu dans une structure de données. Un domaine ne peut qu'être réduit : aucune valeur n'y est jamais ajoutée.

Une *contrainte* impose des restrictions sur les combinaisons de valeurs que peuvent prendre les variables. La *portée* d'une contrainte est le sous-ensemble des variables sur lesquelles elle est définie. Par exemple, la contrainte c :

$$x_2 - 2x_5 + 3x_8 < 4$$

a pour portée :

$$\{x_2, x_5, x_8\}$$

Une contrainte est définie comme un sous-ensemble strict du produit cartésien des domaines des variables de sa portée, restreignant ainsi les combinaisons de

valeurs possibles. Dans l'exemple précédent, supposons que les domaines initiaux des variables sont :

$$D_{x_2} = D_{x_5} = D_{x_8} = \{1, 2\}$$

La contrainte c peut alors être définie par :

$$\{(1,1,1),(1,2,1),(1,2,2),(2,1,1),(2,2,1)\} \subset D_{x_2} \times D_{x_5} \times D_{x_8}$$

pour une permutation lexicographique naturelle $\langle x_2, x_5, x_8 \rangle$ de sa portée. On écrit $(1,1,1) \in c$ puisque cette combinaison de valeurs satisfait la contrainte, mais $(2,2,2) \notin c$. Il s'agit ici d'une définition en extension ; une définition en compréhension, comme $x_2 - 2x_5 + 3x_8 < 4$, est tout aussi acceptable et même souvent préférée.

10.2.2 Problème de satisfaction de contraintes

On appelle *instanciation* d'un ensemble de variables X une application $\mathcal{I}$: $x \in X \mapsto v \in D_x$. On dit que $\mathcal{I}$ *satisfait* une contrainte c si la restriction de $\mathcal{I}$ à la portée de c donne un n-uplet appartenant à c. Les problèmes combinatoires que la PPC s'efforce de résoudre sont généralement définis comme *problèmes de satisfaction de contraintes*, dont voici un exemple.

Étant donné :

- un ensemble fini de variables, $X = \{x_1, \ldots, x_n\}$,
- un domaine fini de valeurs possibles pour chaque variable, $\mathcal{D} = \{D_{x_1}, \ldots, D_{x_n}\}$, $x_i \in D_{x_i} \quad (1 \leq i \leq n)$,
- un ensemble fini de contraintes sur les variables, $C = \{c_1, \ldots, c_m\}$,

trouver une instanciation de X satisfaisant simultanément toutes les contraintes dans C. S'il n'existe pas une telle instanciation, on dit que le problème *n'est pas satisfiable*.

10.2.3 Langage de contraintes

La définition d'une contrainte donnée à l'article 10.2.1 laissait déjà entrevoir sa richesse expressive. La méthode de résolution en PPC, que nous décrivons en détail à la section 10.3, n'exige pas une forme aussi rigide que l'algorithme du simplexe, par exemple, pour les contraintes exprimées dans un modèle. Il suffit qu'on puisse décrire les combinaisons de valeurs permises par la contrainte.

Il demeure quand même qu'une formulation en compréhension des contraintes est privilégiée. Parmi ces contraintes, les contraintes arithmétiques peuvent être construites à partir des opérateurs relationnels habituels ($=$, $<$, $\leq$, $>$, $\geq$), mais aussi à partir de l'opérateur de différence ($\neq$), qui est fréquemment sollicité.

Ces contraintes sont linéaires ou encore non linéaires. Des opérateurs logiques ($\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$) sont également souvent utilisés.

À ces différents types de contraintes s'ajoutent d'autres contraintes, dites globales, exprimant des sous-structures courantes de problèmes combinatoires (art. 10.2.5).

10.2.4 Exemple de modèle

Afin d'illustrer la façon de modéliser en PPC, prenons un problème simplifié d'emploi du temps, celui d'assurer trois quarts de travail différents (notés α, β, γ) par jour sur un horizon d'une semaine (soit sept jours ordonnés : $lu, ma, \ldots, di$) à l'aide de quatre employés (A, B, C, D) travaillant au plus un quart par jour. Voici un modèle possible en PPC :

$$e_{ij} \neq e_{kj} \qquad lu \leq j \leq di, \quad \alpha \leq i < k \leq \gamma \tag{10.1}$$

$$e_{ij} \in \{A, B, C, D\} \qquad lu \leq j \leq di, \quad \alpha \leq i \leq \gamma \tag{10.2}$$

Selon la contrainte (10.2), chaque variable e_{ij} prend pour valeur l'employé qui assurera le quart i au jour j. En comparaison, les formulations classiques de programmation en nombres entiers auraient défini une variable de décision *binaire* pour chaque paire employé-quart ou employé-trame (chap. 4). La contrainte (10.1) évite que deux quarts de travail d'un même jour soient assurés par le même employé. Restreignons davantage le problème en exigeant que chaque employé travaille au moins cinq quarts pendant la semaine :

$$t_{jk} = 1 \leftrightarrow \bigvee_{\alpha \leq i \leq \gamma} e_{ij} = k \qquad lu \leq j \leq di, \quad A \leq k \leq D \tag{10.3}$$

$$\sum_{lu \leq j \leq di} t_{jk} \geq 5 \qquad A \leq k \leq D \tag{10.4}$$

$$t_{jk} \in \{0, 1\} \qquad lu \leq j \leq di, \quad A \leq k \leq D \tag{10.5}$$

Pour ce faire, on définit les variables 0-1 t_{jk} de la contrainte (10.5) [1] et on les lie au fait que l'employé k travaille le jour j selon la contrainte (10.3). La contrainte (10.4) assure que chaque employé travaille suffisamment.

Ce problème demeure encore très simple à résoudre. Son intérêt réside dans sa capacité d'illustrer plusieurs des notions importantes en PPC. Nous le reprendrons tout au long de ce chapitre et ajouterons à sa complexité. Remarquons déjà la forme particulière du modèle : des variables discrètes prenant une valeur

[1] Nous verrons à la section 10.2.5 qu'il y a une meilleure façon de modéliser une telle restriction sans avoir recours à des variables 0-1.

parmi un grand nombre de possibilités (10.2), des contraintes de différence (10.1) et des opérateurs logiques qu'on exprime directement (10.3).

10.2.5 Quelques contraintes globales

Cette section présente quelques contraintes globales parmi les plus usuelles et notamment celles que le lecteur trouvera dans les autres chapitres de ce livre.

La *contrainte d'exclusion mutuelle* pour un ensemble de variables X, généralement notée `alldifferent`(X), impose que les valeurs prises par chacune des variables de l'ensemble soient toutes distinctes.

Par exemple, on pourrait remplacer la contrainte (10.1) pour le problème d'emploi du temps par :

$$\texttt{alldifferent}((e_{ij})_{\alpha \leq i \leq \gamma}) \qquad lu \leq j \leq di \tag{10.6}$$

La *contrainte de distribution des valeurs* pour un ensemble de variables X, également appelée contrainte de cardinalité généralisée (gcc) et souvent notée `distribute`(Y, V, X), lie la suite de variables $Y = \langle y_1, y_2, \ldots, y_m \rangle$ au nombre de répétitions de chacune des valeurs de la suite $V = \langle v_1, v_2, \ldots, v_m \rangle$ prises par les variables de X. Plus précisément, la variable y_i a pour valeur le nombre de fois qu'une variable de X prend la valeur v_i. La contrainte d'exclusion mutuelle `alldifferent` correspond en somme au cas particulier où $y_i \leq 1$ $(1 \leq i \leq m)$.

Cette nouvelle contrainte permet d'exprimer plus directement, et surtout plus globalement que par les contraintes (10.3)–(10.5), la restriction de l'exemple sur le nombre de jours travaillés par chaque employé :

$$\texttt{distribute}(\langle t_A, t_B, t_C, t_D \rangle, \langle A, B, C, D \rangle, (e_{ij})_{lu \leq j \leq di, \alpha \leq i \leq \gamma}) \tag{10.7}$$

$$t_k \in \{5, 6, 7\} \quad A \leq k \leq D \tag{10.8}$$

On remarque qu'ici, t_k ne peut pas prendre la valeur 7 puisqu'on a 21 quarts assurés par 4 employés en prenant chacun au moins 5, pour un minimum de 20. L'article 10.3.3 traite des algorithmes de filtrage associés à la contrainte `distribute` qui sauraient retirer cette valeur.

La *contrainte de relation fonctionnelle* entre deux variables, qu'on note `element`(x, V, y), fait correspondre la variable x à la valeur indexée par la variable y dans la suite $V = \langle v_1, v_2, \ldots, v_m \rangle$. Plus précisément, la variable x a pour valeur le y^{e} élément de V, v_y.

Le problème d'emploi du temps était jusqu'à maintenant un problème de satisfaction et non d'optimisation. Supposons, pour les besoins de l'illustration, que

le coût de couverture[2] d'un certain quart de travail varie selon l'employé qui l'assure et que l'on veuille minimiser la somme des coûts pour assurer le quart β sachant qu'ils sont de 3, 1, 3 et 2 respectivement pour les employés A, B, C et D. On peut écrire :

$$\min \sum_{lu \leq j \leq di} c_j \tag{10.9}$$

$$\texttt{element}(c_j, \langle 3,1,3,2 \rangle, e_{\beta j}) \qquad lu \leq j \leq di \tag{10.10}$$

La fonction objectif est une somme de variables c_j représentant le coût associé au quart β chaque jour de la semaine tandis que la contrainte (10.10) lie ces variables de coût aux variables de décision correspondantes.

La *contrainte d'ordonnancement avec ressource cumulative*, aussi appelée contrainte de mise en boîte et notée $\texttt{cumulative}(S, P, R, k)$, agit sur la suite de variables $S = \langle s_1, s_2, \ldots, s_n \rangle$ représentant le moment où débute chacune de n tâches de durée respective $P = \langle p_1, p_2, \ldots, p_n \rangle$ et de consommation respective de ressource $R = \langle r_1, r_2, \ldots, r_n \rangle$. Étant donné une quantité k de ressource disponible à tout moment, les tâches doivent être planifiées de telle façon qu'on ne dépasse jamais cette capacité.

La *contrainte de longueur de bloc* pour une suite $X = \langle x_1, x_2, \ldots, x_n \rangle$ de variables, notée $\texttt{stretch}(X, L^{\min}, L^{\max})$, contraint le nombre de variables consécutives de même valeur dans X. Soit $V = \{v_1, v_2, \ldots, v_m\}$, l'ensemble des valeurs pouvant être prises par les variables de X et appelons v_i-*bloc*, toute sous-suite maximale de variables prenant la valeur v_i dans X. Les suites d'entiers $L^{\min} = \langle \ell_1^{\min}, \ell_2^{\min}, \ldots, \ell_m^{\min} \rangle$ et $L^{\max} = \langle \ell_1^{\max}, \ell_2^{\max}, \ldots, \ell_m^{\max} \rangle$ précisent, pour chaque valeur v_i, la longueur minimum $\ell_i^{\min}$ et maximum $\ell_i^{\max}$ de tout v_i-bloc dans X.

La *contrainte d'appartenance à un langage régulier* pour une suite de variables $X = \langle x_1, x_2, \ldots, x_n \rangle$, notée $\texttt{regular}(X, \mathcal{A})$, impose que la suite de valeurs prises par les variables de X corresponde à un mot du langage régulier décrit par l'automate fini déterministe $\mathcal{A}$.

La figure 10.1 donne un exemple d'automate correspondant à un patron courant pour des horaires de personnel. Chaque état est représenté par un cercle, le plus à gauche étant l'état initial. Chaque arc marque une transition possible, son étiquette représentant la valeur dont la présence permet cette transition.

[2] On dit qu'un employé *couvre* un quart de travail.

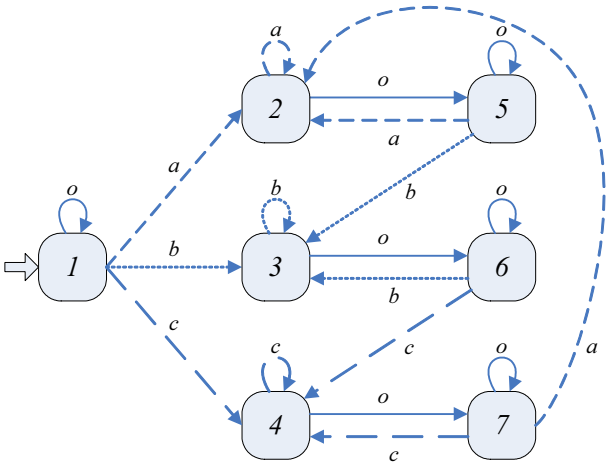

Figure 10.1 Automate correspondant à un patron courant pour des horaires de personnel.

Une telle contrainte sert typiquement à préciser des patrons de valeurs pour X, bien que sa puissance expressive dépasse ce cadre : par exemple, la contrainte de longueur de bloc décrite précédemment en est un cas particulier.

10.2.6 Conseils de modélisation

Pour terminer cette section, voici quelques conseils utiles pour la modélisation.

10.2.6.1 Choisir les variables de décision

La première étape de modélisation consiste à définir les variables. En règle générale, il faut chercher à réduire le nombre de variables qu'on utilise, et ce, pour deux raisons. Tout d'abord, l'espace de recherche des solutions croît de façon exponentielle avec le nombre de variables. Ensuite, la taille des domaines est le plus souvent inversement proportionnelle au nombre de variables. Comme nous le verrons à la section 10.4, les domaines des variables sont fort utiles pour guider la résolution du problème : à l'extrême, des domaines ne contenant que deux valeurs donnent très peu d'information.

Par conséquent, le choix de variables 0-1 est rarement heureux, pour les raisons que nous venons d'évoquer. Ce conseil est d'autant plus important pour le lecteur familier avec la programmation en nombres entiers et pour qui la modélisation en variables 0-1 est peut-être un réflexe. Prenons par exemple l'affectation d'un élément d'un ensemble J à chacun des éléments d'un autre ensemble I. On aurait

comme une modélisation typique de programmation en nombres entiers :

$$\begin{aligned} x_{ij} \in \{0,1\} \qquad & \forall \quad i \in I,\ j \in J \\ \sum_{j \in J} x_{ij} = 1 \qquad & \forall \quad i \in I \end{aligned}$$

où $x_{ij} = 1$ si et seulement si l'élément j est affecté à i, alors qu'en programmation par contraintes on écrirait plutôt :

$$y_i \in J \qquad \forall \quad i \in I$$

où y_i prend pour valeur l'élément de J qui lui est affecté.

Une fois les variables choisies, la définition des domaines suit naturellement.

10.2.6.2 Étudier plusieurs modèles possibles

Les dimensions d'un problème apparaissent dans le modèle sous la forme de domaines des variables et d'indices de celles-ci. Ces rôles sont souvent interchangeables et l'exploration des différentes possibilités permet de faire un choix plus judicieux en fonction des contraintes qu'il faudra exprimer. Par exemple, dans le modèle (10.1),(10.2), ou encore (10.6),(10.2), les employés se retrouvent dans les domaines alors que les jours et les quarts jouent le rôle d'indices des variables de décision. Un modèle dual pouvant le remplacer place les quarts comme domaines (avec la valeur additionnelle δ pour indiquer un congé) et les jours et employés comme indices :

$$\texttt{distribute}(\langle 1,1,1\rangle, \langle \alpha,\beta,\gamma\rangle, (p_{jk})_{A \le k \le D}) \quad lu \le j \le di \tag{10.11}$$

$$p_{jk} \in \{\alpha,\beta,\gamma,\delta\} \quad lu \le j \le di, \quad A \le k \le D \tag{10.12}$$

Ce modèle a l'avantage de pouvoir exprimer plus aisément des contraintes sur les horaires individuels des employés, donnés par la suite de variables $(p_{jk})_{j=lu...di}$ pour un employé k.

10.2.6.3 Utiliser des contraintes globales

Bien que cela demande une certaine dose d'expérience, l'utilisation de contraintes globales mène habituellement à une résolution plus efficace du problème. Comme leur nom l'indique, ces contraintes prennent un point de vue plus global qui se traduit par des raisonnements plus sophistiqués (sect. 10.3). La contrainte (10.6) en est un bon exemple.

10.2.6.4 Ajouter des contraintes redondantes

Supposons dans l'exemple que les durées des quarts de travail peuvent varier selon le type de quart et le jour de la semaine et qu'elles sont données par $(d_{ij})_{lu \leq j \leq di, \alpha \leq i \leq \delta}$. Supposons aussi une semaine de 35 heures précisément pour les employés A, B, D et de 20 heures pour l'employé C, travaillant à temps partiel. On peut modéliser ainsi cette situation :

$$\texttt{element}(h_{jk}, (d_{ij})_{\alpha \leq i \leq \delta}, p_{jk}) \qquad lu \leq j \leq di, \quad A \leq k \leq D \tag{10.13}$$

$$\sum_{lu \leq j \leq di} h_{jk} = 35 \qquad k \in \{A, B, D\} \tag{10.14}$$

$$\sum_{lu \leq j \leq di} h_{jC} = 20 \tag{10.15}$$

Malheureusement, une telle modélisation n'est pas très efficace parce que les contraintes (10.14) et (10.15) implantent généralement la cohérence de bornes (art. 10.3.2). Une option intéressante prend le point de vue d'un problème de mise en boîte où, pour chaque quart de travail, on crée une petite boîte de largeur unitaire et dont la hauteur est égale à sa durée d_{ij}. Ces boîtes doivent toutes être placées dans une plus grande, de largeur quatre (le nombre d'employés) et de hauteur 35 (le nombre d'heures travaillées par semaine). On peut visualiser cette grande boîte comme un ensemble de bandes de largeur unitaire, une par employé (fig. 10.2). La bande dans laquelle se trouve une petite boîte s'interprète alors comme l'employé effectuant le quart de travail correspondant. Pour modéliser le fait que l'employé C ne travaille que 20 heures par semaine, on ajoute une boîte fictive déjà placée dans la bande C et de hauteur 15, laissant une hauteur résiduelle de 20 heures pour les autres tâches.

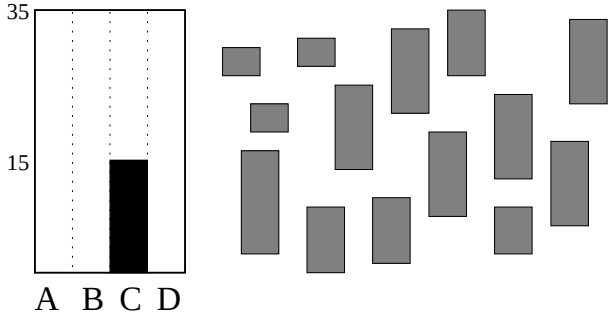

Figure 10.2 Modélisation des charges de travail individuelles comme un problème de mise en boîte. Les boîtes grises représentent les quarts de travail et la boîte noire déjà placée permet de réduire la charge effective de l'employé C.

L'ajout de la contrainte :

$$\texttt{cumulative}(\langle e_{lu\ \alpha}, \ldots, e_{di\ \gamma}, C\rangle, \langle 1, \ldots, 1\rangle, \langle d_{lu\ \alpha}, \ldots, d_{di\ \gamma}, 15\rangle, 35) \quad (10.16)$$

réintroduit les variables e_{ij} et ajoute donc une redondance dans le modèle. Mais parce qu'il exprime d'une façon différente et complémentaire une certaine sous-structure du problème, ce superflu apparent peut mener à une résolution plus efficace.

10.3 PROPAGATION

10.3.1 Filtrage et propagation

Le moteur de résolution à la base de la PPC, appelé *propagation de contraintes*, diffuse à travers l'ensemble des contraintes les résultats de raisonnements locaux à chacune d'entre elles, obtenus par des algorithmes de filtrage. Chaque type de contrainte possède son propre *algorithme de filtrage* pour éliminer du domaine des variables de sa portée les valeurs localement incohérentes, ne pouvant pas mener à une solution. La communication entre les différents algorithmes de filtrage se fait à travers les domaines des variables. L'algorithme de filtrage d'une contrainte pourra s'activer lorsque le domaine d'une des variables de sa portée est modifié. Ce mécanisme facilite grandement la coopération de contraintes hétéroclites et explique la flexibilité dans la formulation des modèles.

Il est préférable pour certains types de contraintes de ne pas activer leur algorithme de filtrage chaque fois qu'un domaine pertinent est modifié. Parfois, il est suffisant de ne l'activer que lorsque ce domaine ne contient plus qu'une seule valeur ou encore lorsque la plus petite ou la plus grande valeur du domaine est retirée (art. 10.3.3). Bien qu'un algorithme de filtrage, en réduisant un domaine, puisse en activer un autre et ainsi de suite, la propagation de contraintes se termine nécessairement puisque chaque activation est le résultat du retrait d'une valeur dans un domaine et que nous ne pouvons qu'en retirer un nombre fini de fois : en effet, le nombre de variables dans un modèle est fini et chaque domaine est également fini. La réduction d'un seul domaine provoque parfois l'activation de plusieurs algorithmes de filtrage. Une autre propriété importante est que le résultat final de la propagation ne dépend pas de l'ordre dans lequel ceux-ci sont activés.

10.3.2 Caractérisation du niveau de cohérence

Bien qu'il soit certainement profitable qu'un algorithme de filtrage élimine des valeurs incohérentes du domaine des variables, il est également souhaitable de

caractériser les valeurs à éliminer. On utilise la notion de *niveau de cohérence*[3] comme mesure de qualité des algorithmes de filtrage. Du point de vue théorique, plusieurs niveaux de cohérence ont été définis et ceux-ci servent à décrire le travail effectué par un algorithme de filtrage. Historiquement, on a d'abord appliqué la notion de cohérence aux contraintes binaires.

Définition 10.1 *Étant donné une contrainte binaire c de portée $\{x, y\}$ et les domaines D_x et D_y pour les variables x et y respectivement, la* ***cohérence d'arc*** *est atteinte pour c si : et seulement si*

- *pour chaque valeur $v \in D_x$, il existe $(v, v') \in c$ où $v' \in D_y$ et*
- *pour chaque valeur $v \in D_y$, il existe $(v', v) \in c$ où $v' \in D_x$.*

La cohérence d'arc garantit que les domaines des deux variables ne contiennent que des valeurs pouvant mener à une solution pour la contrainte : chaque valeur pour une variable est supportée par au moins une autre pour l'autre variable (on dit qu'elle constitue son *support*). On peut étendre cette notion de cohérence, comme toutes les autres d'ailleurs, à l'ensemble des contraintes d'un problème de satisfaction de contraintes $\mathcal{P}$. Si chacune des contraintes atteint la cohérence d'arc, on dit que la cohérence d'arc est atteinte pour $\mathcal{P}$. Notons que cela ne veut pas dire que toute valeur restante participe nécessairement à une solution de $\mathcal{P}$: la propriété de cohérence demeure locale à chacune des contraintes.

Cette notion de cohérence se généralise facilement aux contraintes mettant en jeu plus de deux variables. Selon les auteurs, on parlera tantôt de cohérence d'hyper-arc, tantôt de cohérence d'arc généralisée. Nous préférerons l'expression cohérence de domaine.

Définition 10.2 *Étant donné une contrainte c de portée $\{x_1, x_2, \ldots, x_k\}$ et les domaines D_{x_i} $(1 \leq i \leq k)$, la* ***cohérence de domaine*** *est atteinte pour c si et seulement si pour chaque x_i $(1 \leq i \leq k)$ et chaque valeur $v \in D_{x_i}$, il existe $(v_1, \ldots, v_{i-1}, v, v_{i+1}, \ldots, v_k) \in c$ où $v_j \in D_{x_j}$.*

En général, atteindre la cohérence de domaine pour une contrainte demande un effort qui croît de façon exponentielle avec le nombre de variables dans sa portée[4]. La cohérence de bornes est une option plus abordable, quoique strictement moins forte. Elle s'appuie sur une relaxation des domaines discrets des variables à des intervalles continus. On note $x^{\min}$ et $x^{\max}$ les plus petite et plus grande valeurs dans D_x, respectivement.

[3] Le terme « consistance » est également utilisé pour sa proximité lexicale (mais malheureusement pas sémantique) de l'anglais *consistency*.

[4] Nous verrons néanmoins à l'article 10.3.3 que la structure sous-jacente peut parfois être exploitée pour que l'effort demeure polynomial.

Définition 10.3 *Étant donné une contrainte c de portée* $\{x_1, x_2, \ldots, x_k\}$ *et les domaines* D_{x_i} $(1 \leq i \leq k)$*, la* ***cohérence de bornes*** *est atteinte pour c si et seulement si pour chaque* x_i $(1 \leq i \leq k)$ *:*

- *il existe* $(r_1, \ldots, r_{i-1}, x_i^{\min}, r_{i+1}, \ldots, r_k) \in c$ *où* r_j *est un nombre réel appartenant à l'intervalle* $[x_j^{\min}, x_j^{\max}]$ *et*
- *il existe* $(r_1, \ldots, r_{i-1}, x_i^{\max}, r_{i+1}, \ldots, r_k) \in c$ *où* r_j *est un nombre réel appartenant à l'intervalle* $[x_j^{\min}, x_j^{\max}]$*.*

10.3.3 Algorithmes de filtrage

Les algorithmes de filtrage peuvent être classés en deux catégories, selon le type de contrainte pour lequel ils sont définis. Les contraintes *simples* ont généralement des algorithmes de filtrage relativement faciles à définir alors que les contraintes *globales* font appel à des techniques plus complexes, parfois issues de la recherche opérationnelle.

10.3.3.1 Contraintes binaires

Le développement d'algorithmes de filtrage des contraintes binaires a constitué un domaine de recherche extrêmement actif au cours des dernières années. Les algorithmes, qui permettent d'atteindre la cohérence de domaine, sont notés par les lettres AC (*Arc Consistency*) suivies d'un numéro les identifiant. Ainsi, une dizaine d'algorithmes (AC-3, AC-4, AC-5, AC-6, AC-7, AC-2000, AC-2001, AC-8, AC-3_d, AC-3.2 et AC-3.3) se distinguent par leur complexité en temps et en espace mémoire utilisé. Nous donnons ici les grandes lignes de quelques-unes de ces techniques puisque plusieurs sont des variations d'une même idée. Les mesures de complexité concernent une contrainte binaire liant, par exemple, x et y.

AC-3 est l'algorithme de base qui applique directement la définition de la cohérence de domaine, c'est-à-dire qu'il vérifie que pour chaque valeur a de D_x, il existe bien une valeur b dans D_y agissant comme support aux yeux de la contrainte pour laquelle on filtre. La complexité en espace est donc nulle ; par contre, la complexité en temps est dans $O(d^3)$, où d est la taille du plus grand domaine.

AC-4, pour accélérer le temps de filtrage, propose de conserver dans un tableau la liste de toutes les solutions partielles (en regard de x et de y) associées à chaque valeur a de D_x. En implantant correctement cette structure de données, on peut réduire la complexité en temps à $O(d^2)$, mais la complexité en espace atteint elle aussi $O(d^2)$.

AC-6 et AC-7 poussent un peu plus loin cette idée en montrant qu'il est possible de ne conserver qu'un seul support pour chaque valeur. Pour chaque paire variable-valeur (x, a), ces algorithmes maintiennent la liste de toutes les autres paires variable-valeur supportées par (x, a). Ainsi, on conserve une complexité en temps de $O(d^2)$ mais la complexité en espace est réduite à $O(d)$.

Les autres techniques proposent des raffinements ainsi que la combinaison et l'amélioration des techniques vues précédemment.

10.3.3.2 Contraintes arithmétiques

Dans le cas des contraintes arithmétiques, leur sémantique permet de spécialiser et d'accélérer les algorithmes de filtrage. Comme auparavant, soit $x^{\max}$, la valeur maximum du domaine d'une variable x et $x^{\min}$, sa valeur minimum. Le filtrage des contraintes arithmétiques telles que $=, \neq, <, \leq, >$ et $\geq$ liant deux variables x et y s'effectue par l'application des traitements suivants :

- $x = y$: lorsqu'une des deux variables prend une valeur $(x = v)$, la même valeur est attribuée à la seconde $(y = v)$. Réciproquement, lorsqu'une valeur est retirée du domaine d'une des variables $(v \notin D_x)$, elle est aussi retirée du domaine de la seconde $(v \notin D_y)$.
- $x \neq y$: lorsqu'une des deux variables prend une valeur $(x = v)$, cette valeur est alors retirée du domaine de l'autre variable $(v \notin D_y)$.
- $x \leq y$ $(x < y)$: le filtrage de cette contrainte consiste à éliminer toute valeur v de D_x qui soit supérieure (ou égale) à $y^{\max}$. De la même manière, on doit retirer toute valeur v de D_y qui soit inférieure (ou égale) à $x^{\min}$. Le traitement pour $x \geq y$ $(x > y)$ est très semblable.

Par exemple, si on considère les deux variables x et y telles que $D_x = \{1, 2, 3, 4\}$ et $D_y = \{3, 4, 5\}$ ainsi que la contrainte $x > y$, le filtrage associé permet de conclure que $x = 4$ et $y = 3$. Il est à noter que toutes ces méthodes de filtrage pour les contraintes arithmétiques binaires atteignent la cohérence de domaine.

Les contraintes arithmétiques non binaires ont généralement la forme $a_1x_1 + a_2x_2 + \cdots + a_nx_n = d$ où $a_1, \ldots, a_n$ sont des constantes, $x_1, \ldots, x_n$ des variables, d est une constante et où l'opérateur de comparaison peut-être substitué par $<, \leq, >$, ou $\geq$. Pour effectuer le filtrage de ces contraintes, on se contente généralement d'atteindre la cohérence de borne en raisonnant sur les valeurs extrêmes de chaque variable. Par exemple, si on veut filtrer la contrainte $2x - y - 3z \leq 5$ pour la variable y, on déduit que $y \geq 2x - 3z - 5$ et donc qu'il est possible d'éliminer du domaine (D_y) de y toutes les valeurs inférieures à $2x^{\min} - 3z^{\max} - 5$. On procède ensuite de la même manière pour x et z en éliminant de D_x toutes les valeurs supérieures à $(y^{\max} + 3z^{\max} + 5)/2$ et en retirant de D_z toutes les valeurs inférieures à $(2x^{\min} - y^{\max} - 5)/3$.

10.3.3.3 Contraintes globales

Nous avons vu dans l'article10.2.5 quelques contraintes globales fréquemment utilisées en programmation par contraintes. Nous donnons ici quelques détails sur les algorithmes de filtrage associés à la contrainte de distribution de valeurs, à la contrainte d'ordonnancement avec ressource cumulative et à la contrainte d'appartenance à un langage régulier.

Contraintes de distribution de valeurs . La contrainte `distribute`(Y, V, X) peut être représentée par un ensemble de contraintes plus simples de la manière suivante :

$$y_j = \sum_{i \in \{1,2,\ldots,n\}} (x_i = v_j) \qquad j \in \{1, 2, \ldots, m\}$$

Si on considère de manière simultanée l'ensemble des variables Y, il est possible d'obtenir un filtrage plus efficace. L'algorithme de filtrage proposé par Régin [4] est le plus couramment utilisé pour atteindre et maintenir la cohérence de domaine pour cette contrainte. Cette technique est basée sur un algorithme de flot appliqué à un graphe particulier nommé le graphe de valeurs $G = (N, A)$. Ce graphe (fig. 10.3) est constitué de deux couches de nœuds, l'une représentant les variables et l'autre, l'ensemble des valeurs. Un arc (i, j) est défini entre une paire de nœuds de types différents (une variable x_i et une valeur v_j) lorsque la valeur v_j appartient au domaine de la variable x_i.

On ajoute ensuite à ce graphe deux nœuds additionnels : s lié à tous les nœuds de type variable et t relié à tous les nœuds de type valeur. À chaque arc (i, j) du graphe, on associe une paire de valeurs (d_{ij}, c_{ij}) où d_{ij} représente la demande minimum devant le traverser et c_{ij} exprime sa capacité maximale. Tous les arcs reliant s aux nœuds de variables se voient attribuer une demande et une capacité de 1, (d_{si}, c_{si}) = (1,1), alors que les arcs reliant les variables aux valeurs de leur domaine sont marqués d'une demande de 0 et d'une capacité de 1, (d_{ij}, c_{ij}) = (0,1). On attribue ensuite aux arcs reliant une valeur v_j au nœud t une demande égale à la valeur minimum du domaine de y_j et une capacité équivalente à la valeur maximale du domaine de y_j, $(d_{jt}, c_{jt}) = (y_j^{\min}, y_j^{\max})$. Enfin, on associe à l'arc reliant t à s la paire (n,n), où n correspond au nombre de variables de X.

Pour vérifier si la contrainte est cohérente par rapport aux domaines des variables X, il suffit de définir un flot réalisable dans le graphe des valeurs (fig. 10.3), soit de faire circuler n unités de flot de s vers t. Définir un flot consiste à attribuer

à chaque arc (i, j) une valeur f_{ij} de manière à ce que :

$$d_{ij} \leq f_{ij} \leq c_{ij} \qquad \forall\, (i,j) \in A$$
$$\sum_{i|(i,k)\in A} f_{ik} = \sum_{j|(k,j)\in A} f_{kj} \qquad \forall\, k \in N$$

c'est-à-dire que le flot de chaque arc soit compris entre sa demande et sa capacité, et ce, tout en respectant la conservation du flot à chaque nœud.

Il est ensuite possible de maintenir la cohérence de domaine de manière très efficace en travaillant avec les composantes fortement connexes du graphe résiduel associé au flot. Le lecteur peut consulter Régin [4] pour connaître les détails de cette technique.

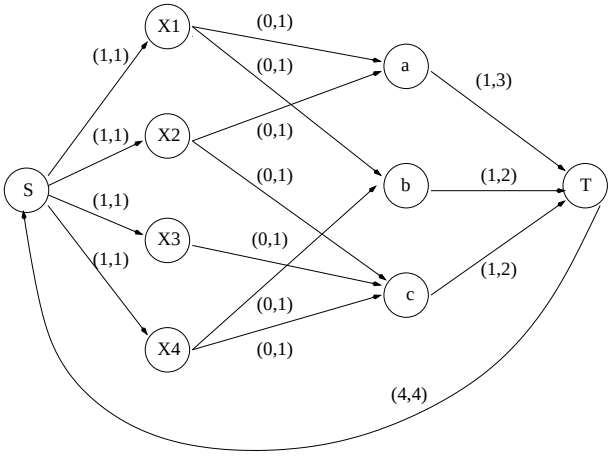

Figure 10.3 Graphe des valeurs pour le filtrage de `distribute`(Y, V, X) où $Y = \{y_a, y_b, y_c\}$, $D_Y = \{\{1,2,3\},\{1,2\},\{1,2\}\}$, $V = \{a, b, c\}$ et $X = \{x_1, x_2, x_3, x_4\}$, $D_X = \{\{a, b\},\{a, c\},\{c\},\{b, c\}\}$.

Contraintes d'ordonnancement avec ressource cumulative. Pour la contrainte `cumulative`(S, P, R, k), on a proposé plusieurs algorithmes de filtrage (atteignant la cohérence de borne). Les algorithmes principaux sont basés sur le calcul de bilans énergétiques. L'énergie utilisée par une tâche i durant son exécution est donnée par la valeur $e_i = r_i p_i$, produit de la durée de la tâche par la ressource consommée. Nous présentons ici une première règle permettant de déterminer par raisonnement énergétique si un problème est irréalisable.

On peut estimer durée minimale $p_i(t_1, t_2)$ d'exécution d'une tâche i sur un intervalle de temps $[t_1, t_2]$ à partir des dates possibles d'exécution de i, à savoir les intervalles de temps $[s_i, s_i + p_i]$ tels que $s_i \in \{s_i^{\min}, \ldots, s_i^{\max}\}$ (où $s_i^{\min}$ et $s_i^{\max}$ représentent les dates au plus tôt et au plus tard de début de la tâche i).

Une estimation provient de $p_i(t_1, t_2) = \max(0, \min(p_i, t_2 - t_1, s_i^{\min} + p_i - t_1, t_2 - s_i^{\max}))$ et l'énergie consommée par i dans l'intervalle $[t_1, t_2]$ est supérieure ou égale à $e_i(t_1, t_2) = r_i \cdot p_i(t_1,_t 2)$. L'énergie totale minimale consommée par les tâches est donnée par $\sum_{i=1,2,\ldots,n} e_i(t_1, t_2)$. Elle doit être au plus égale à l'énergie totale disponible $k(t_2 - t_1)$, sinon le problème est irréalisable. Les valeurs de t_1 et t_2 sont choisies parmi les temps de début au plus tôt ($s^{\min}$) et au plus tard ($s^{\max}$) de toutes les tâches.

Baptiste *et al.* [5] montrent comment le raisonnement énergétique permet aussi d'effectuer des ajustements des bornes $s_i^{\min}$ et $s_i^{\max}$ en $O(n^3)$, ainsi que d'autres règles de filtrage associées à la contrainte cumulative. Par une étude expérimentale, les auteurs ont aussi mis à jour des dominances entre différentes règles d'ajustement applicables au RCPSP (chap. 5). Ils montrent notamment que le comportement du raisonnement énergétique dépend clairement du type de problème testé. En effet, son coût en temps de calcul se ressent fortement sur les problèmes hautement disjonctifs où un grand nombre de tâches sont incompatibles deux à deux. Au contraire, il prouve son efficacité sur les problèmes hautement cumulatifs.

Toutefois, dans le cas particulier où l'on ne peut exécuter qu'une seule tâche à la fois ($k = 1$), il est généralement plus efficace d'utiliser l'algorithme du *Edge Finding* (chap. 5) pour effectuer le filtrage de la contrainte `cumulative`.

Contraintes d'appartenance à un langage régulier. L'algorithme de filtrage pour la contrainte `regular`$(X, \mathcal{A})$ parcourt la suite de variables $X = \langle x_1, x_2, \ldots, x_n \rangle$ en appliquant l'automate $\mathcal{A}$, bâtissant ainsi un graphe multiparti $(N^1, N^2, \ldots, N^{n+1}, A)$ où chaque partie N^i contient un sommet par état de l'automate et chaque arc de A correspond à une paire variable-valeur (x_i, v_j) [6]. Soit τ, la fonction de transition de l'automate. Dans un premier temps, on ajoute un arc (i, j) allant du nœud correspondant à l'état k dans la partie N^i au nœud correspondant à l'état ℓ dans la partie suivante (N^{i+1}) s'il existe une valeur $v_j \in D_i$ telle que $\tau(k, v_j) = \ell$. Dans un second temps, on retire tout arc n'appartenant pas à un chemin de l'état initial dans la partie N^1 à un état final dans la partie N^{n+1}. Un tel chemin agit comme support des arcs (autrement dit, des paires variable-valeur) le composant. Toute valeur $v_j \in D_{x_i}$ pour laquelle aucun arc (i, j) n'est présent dans le graphe se voit retirée du domaine puisqu'elle n'a pas de support. La figure 10.4 fournit un exemple. Puisque aucun arc $(2, b)$ ou $(4, b)$ ne subsiste, le filtrage initial retire la valeur b du domaine des deuxième et quatrième variables. Le graphe est mis à jour au fur et à mesure des modifications des domaines des variables. L'algorithme résultant assure la cohérence de domaine. Son temps de calcul est linéaire dans la taille du graphe pour l'initialisation et linéaire dans la taille des mises à jour du graphe pour les appels subséquents.

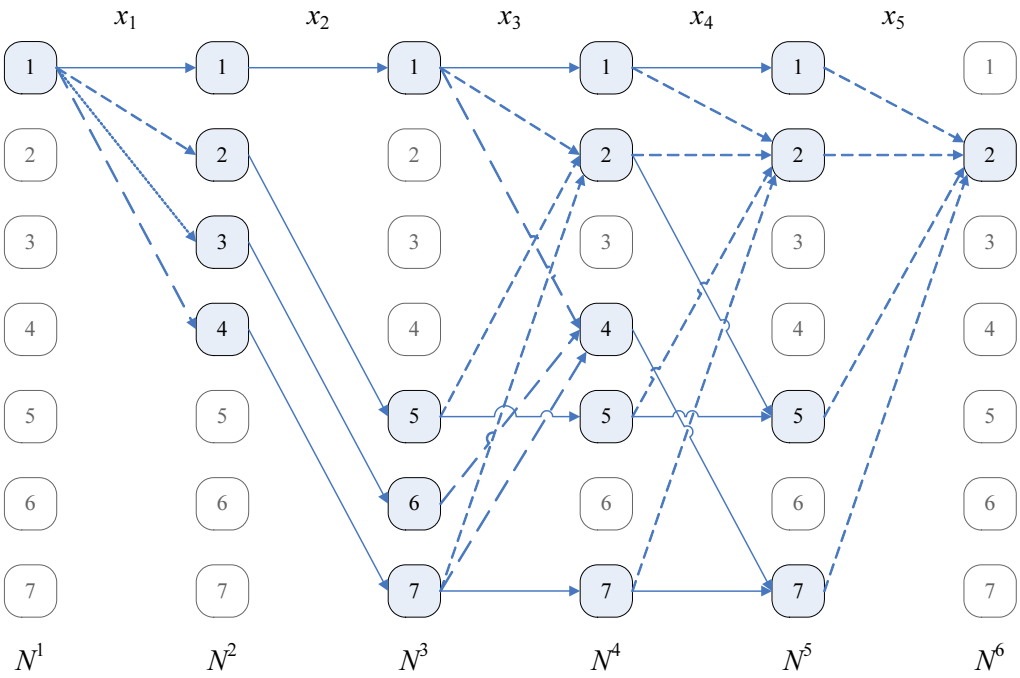

Figure 10.4 Graphe construit à partir de l'automate de la figure 10.1 et d'une suite de cinq variables avec domaines initiaux respectifs $\{a, b, c, o\}$, $\{b, o\}$, $\{a, c, o\}$, $\{a, b, o\}$ et $\{a\}$.

10.4 RECHERCHE

Une fois la propagation des contraintes terminée, il n'est pas garanti que toutes les variables auront été instanciées : certains domaines peuvent encore contenir plusieurs valeurs. Puisque la propagation n'est en pratique jamais suffisante pour déterminer des solutions réalisables, il est généralement nécessaire d'entreprendre une recherche de ces solutions. Ce processus s'effectue par l'ajout dynamique de contraintes et la génération d'un arbre de recherche où chaque feuille représente une solution réalisable.

Les étapes de branchement à chaque nœud N de l'arbre de recherche peuvent se résumer comme suit (le nœud racine n'étant en fait que le problème lui-même) :

1. effectuer la propagation des contraintes de N. Si le domaine d'une variable se vide, alors on rejette N et on *sélectionne* le prochain nœud ;
2. *choisir* une variable x non instanciée ($|D_x| > 1$) ; si toutes les variables sont instanciées, alors on a une solution réalisable ;

3. *choisir* une valeur $v \in D_x$ possible pour x ;
4. créer deux nouveaux nœuds, N' et N'', auxquels on ajoute de nouvelles contraintes à l'ensemble des contraintes du nœud courant $C(N)$:
   (a) $C(N') = C(N) \cup (x = v)$
   (b) $C(N'') = C(N) \cup (x \neq v)$
5. ajouter N' et N'' à la liste des nœuds non traités.

Dans cette procédure, on observe trois degrés de liberté (en italique) qui permettent d'adapter la recherche de solutions à la structure particulière du problème. Ces points d'intervention sont le choix de variable, le choix de valeur et la sélection du prochain nœud à traiter. Avant d'aborder ces choix plus en détail, mentionnons qu'il est possible d'effectuer la recherche de solutions en fragmentant le domaine des variables plutôt qu'en leur assignant une valeur. Pour ce faire, on remplace les contraintes de branchement ($(x = v)$ et $(x \neq v)$) par des contraintes permettant de fractionner le domaine d'une variable en deux ensembles complémentaires.

10.4.1 Heuristiques de choix de variable

Lorsque plusieurs variables restent non instanciées, il est nécessaire de choisir la prochaine variable qui fera l'objet d'un branchement. Cette décision a une incidence majeure sur les performances d'une stratégie de recherche et il est important d'effectuer un choix judicieux.

10.4.1.1 Ordre statique opposé à ordre dynamique.

Lorsque l'ordre de sélection des variables est déterminé avant le début de la recherche, on parle d'un ordre *statique* alors qu'un ordre *dynamique* sous-entend que les variables sont sélectionnées à chaque nœud en fonction du contexte courant et d'un critère donné.

Un ordonnancement statique permet d'accélérer le processus de sélection de variables puisque les calculs ne sont effectués qu'une seule fois, avant le début de l'exploration. Toutefois, cette stratégie ne tire pas profit de l'information disponible et mise à jour durant la recherche. L'ordonnancement dynamique, bien qu'un peu plus coûteux en temps de calcul, permet de prendre de meilleures décisions en exploitant l'information introduite par les nœuds explorés précédemment. Parce qu'il est beaucoup plus performant, on utilise presque toujours l'ordre dynamique.

10.4.1.2 Échec en premier.

Dans le choix de la prochaine variable à instancier, on tente de sélectionner des candidats qui mènent à une impasse ; c'est ce qu'on appelle la stratégie de l'*échec en premier*. L'idée derrière ce principe est de tenter de résoudre le plus rapidement possible les aspects les plus difficiles du problème à traiter. En effet, si une variable est difficile à instancier, cette difficulté ne fera qu'augmenter au fur et à mesure que d'autres variables seront fixées. Il est donc primordial de traiter les variables problématiques le plus tôt possible.

Il existe plusieurs mesures de la difficulté d'instanciation de chaque variable, mais la plus utilisée est sans contredit la taille de son domaine. En effet, le domaine d'une variable qui contient peu de valeurs risque de se vider plus rapidement qu'un domaine qui en contient encore plusieurs. Ainsi, en fixant d'abord les variables qui ont de petits domaines, on réduit les chances qu'elles créent des impasses plus tard.

Ce critère est toutefois général et, lorsqu'on connaît bien la structure d'un problème particulier, il est possible de concevoir une stratégie adaptée qui soit plus performante. Ainsi, dans l'exemple d'emploi du temps examiné dans ce chapitre, on pourrait choisir comme stratégie de sélectionner en premier les variables e_{ij} ou p_{jk} ayant le plus petit domaine. Si la définition du problème était plus riche, on pourrait aussi utiliser d'autres données telles la compétence des employés ou la difficulté à combler certains quarts de travail comme les week-ends par exemple.

10.4.1.3 Problème d'optimisation.

Lorsqu'on résout un problème d'optimisation et qu'on dispose d'un coût c_{xv} associé à chaque paire variable-valeur (x, v), il est aussi possible de sélectionner les variables pour tenter d'obtenir des solutions de bonne qualité. Les critères de coût minimum et de regret maximum sont les plus fréquemment utilisés.

Le critère de coût minimum consiste à choisir la variable x associée au $c^{\star}_{xv}$, le coût minimum pour toutes les paires variable-valeur encore possibles. Dans l'exemple d'emploi du temps, on choisirait donc la variable e_{ij} dont le domaine contient encore l'employé minimisant le coût de couverture de la tâche j. Cette méthode s'avère parfois efficace, mais elle est généralement trop myope (elle considère trop peu d'informations) pour donner de bons résultats.

Le critère de regret maximum, un peu comme celui de l'échec en premier, tente de fixer les variables pour lesquelles l'obtention d'un bon coût est difficile. Cela se mesure avec le regret, soit la différence de coût entre les deux meilleures valeurs pour x. Ce critère permet de traiter rapidement les variables pour lesquelles

la détérioration de l'objectif sera importante s'il devient impossible de leur attribuer la meilleure valeur de leur domaine.

10.4.2 Heuristiques de choix de valeur

Une fois la variable à traiter définie, la prochaine étape consiste à choisir une valeur dans son domaine. Ce choix est moins crucial que le choix de variable et l'approche choisie est contraire à celle présentée à l'article 10.4.1. En effet, ici on essaie plutôt de maximiser les chances de réussite et la qualité de la solution produite. On choisit donc la meilleure valeur présente dans le domaine de la variable choisie.

La meilleure valeur possible peut être définie comme celle qui maximise les chances de satisfaction dans le cas d'un problème de satisfaction de contraintes ou celle qui optimise une fonction objectif dans le cas d'un problème d'optimisation.

Si, dans notre exemple, la contrainte la plus contraignante est l'attribution d'un minimum d'heures de travail hebdomadaire à chaque employé, on pourrait sélectionner l'employé à qui on a attribué le moins d'heures parmi ceux du domaine de la variable e_{ij} choisie.

10.4.3 Exploration de l'arbre

En dernier lieu, il faut déterminer la manière dont sera parcouru l'arbre de recherche, c'est-à-dire dans quel ordre seront développés les nœuds créés par la procédure de branchement.

10.4.3.1 Recherche en profondeur.

On utilise le plus souvent la stratégie de recherche en profondeur (*Depth First Search : DFS*) en programmation par contraintes, et ce, pour plusieurs raisons. D'abord, cette stratégie de recherche est particulièrement bien adaptée à la détermination d'une solution réalisable. En effet, puisque les solutions du problème à résoudre se retrouvent dans les feuilles de l'arbre de recherche, la meilleure stratégie consiste à plonger le plus rapidement possible. Comme la PPC s'est surtout développée autour des problèmes de satisfaction de contraintes, la recherche en profondeur a longtemps été la méthode de référence pour l'exploration arborescente. La figure 10.5 illustre l'ordre dans lequel les feuilles d'un arbre de recherche sont évaluées par une recherche en profondeur. De plus, l'absence d'une méthode de borne universelle (comme la relaxation linéaire en programmation mathématique) rend l'utilisation d'une recherche par meilleur candidat (*Best First Search : BFS*) difficile à implanter dans les librairies de PPC. Enfin, chaque

nœud conservé d'un arbre de recherche en PPC nécessite la sauvegarde de toutes les variables, de tous les domaines et de toutes les contraintes. Le fait que la recherche en profondeur génère peu de nœuds (consommant donc peu de mémoire) a constitué un atout important lors de l'implémentation des premiers solveurs de contraintes.

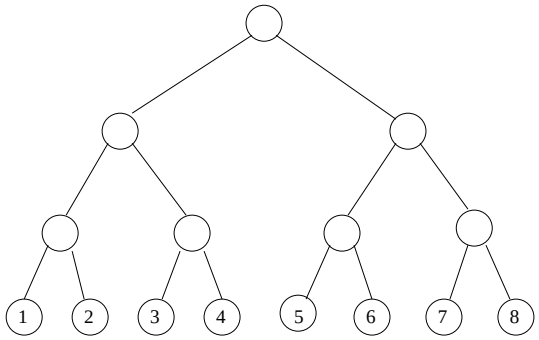

Figure 10.5 Ordre de parcours des feuilles lors d'une recherche en profondeur.

10.4.3.2 Recherche à divergence limitée.

Le principal inconvénient de la recherche en profondeur est qu'elle accorde une très grande importance aux décisions qui sont prises dans les premiers niveaux de l'arbre de recherche. En effet, avant de reconsidérer une mauvaise décision prise dans les premières branches, il est nécessaire d'explorer une sous-arborescence souvent très importante. La recherche à divergence limitée (*Limited Discrepancy Search : LDS*) fut donc proposée pour pallier ce problème et diriger la recherche plus tôt vers de meilleures solutions.

Le principe de base derrière la recherche à divergence limitée est la confiance portée à l'heuristique de choix de valeur. Lorsqu'une branche de l'arbre de recherche suit une affectation variable-valeur qui n'est pas la première suggérée par l'heuristique, on parle alors d'une *divergence*. Le processus de recherche commence donc par traverser l'arbre de recherche en ne permettant aucune divergence (ce qui correspond à toujours suivre la branche de gauche) ; ensuite il tolère une seule divergence, puis deux, trois, etc. De cette manière, les premières solutions établies sont celles qui respectent au mieux les suggestions de l'heuristique de choix de valeur. De plus, une mauvaise décision prise en haut de l'arbre de recherche sera vite annulée lorsque le niveau de divergence permis est bas. On essaie généralement d'effectuer les choix divergents en haut de l'arbre de recherche, puisqu'à ce niveau on dispose de moins d'information et l'heuristique de choix de valeur est plus susceptible de se tromper. La figure 10.6 illustre l'ordre de visite des feuilles lors d'une recherche à divergence limitée.

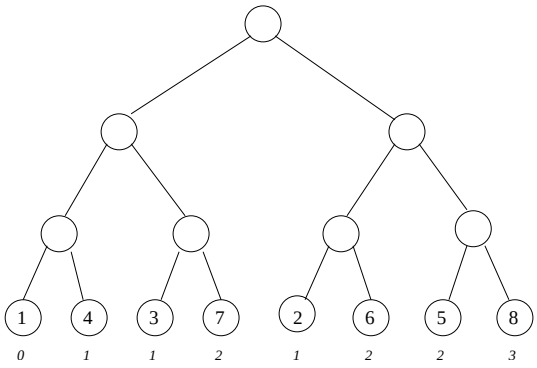

Figure 10.6 Ordre de parcours des feuilles lors d'une recherche à divergence limitée. L'ordre apparaît à l'intérieur du cercle alors que le nombre de divergences est indiqué en dessous.

10.5 APPROCHES HYBRIDES

On nomme généralement *approches hybrides* les méthodes qui utilisent et combinent la programmation par contraintes et d'autres techniques issues de la recherche opérationnelle. Le développement des méthodes hybrides va bon train depuis quelques années et un livre dédié à ce sujet a récemment été publié [7]. Bien que des algorithmes complexes issus de la recherche opérationnelle soient utilisés depuis plusieurs années dans le filtrage des contraintes globales, l'attention des chercheurs a récemment porté sur l'aspect de l'optimisation d'une mesure de qualité. Des méthodes telles que le filtrage basé sur les coûts réduits, la génération de colonnes basée sur la programmation par contraintes, la décomposition de Benders logique et la recherche à grand voisinage illustrent bien cet effort de mise en commun des connaissances et des techniques. Nous survolons ici quelques-unes de ces méthodes qui sont parfois utilisées pour résoudre des problèmes d'ordonnancement ou d'emploi du temps.

L'idée derrière le filtrage basé sur les coûts réduits est d'éliminer du domaine d'une variable non seulement les valeurs menant à des incohérences, mais aussi toutes celles ne menant pas à la solution optimale. Cette méthode est applicable lorsqu'on dispose d'une borne inférieure sur la fonction objectif $o^{\min}$ (pour un problème de minimisation), d'une solution réalisable de valeur z et des coûts réduits c'_{xv} associés au fait que la variable x prenne la valeur v. Pour effectuer un filtrage basé sur les coûts réduits, il suffit d'éliminer de D_x toutes les valeurs v telles que $o^{\min} + c'_{xv} > z$. Cette méthode est particulièrement efficace lorsqu'on dispose d'un algorithme spécialisé pour calculer les coûts réduits.

La génération de colonnes basée sur la programmation par contraintes essaie de tirer profit de la flexibilité de la programmation par contraintes pour résoudre les sous-problèmes associés à la génération de colonnes (chap. 7). Ces problèmes,

qui consistent par exemple à construire une journée de travail, la route d'un véhicule ou le plan de production d'une machine, sont généralement sujets à un grand nombre de contraintes toutes aussi variées que complexes. Le riche langage de la programmation par contraintes permet donc d'exprimer ces contraintes naturellement, et ce, peu importe leur structure. Les problèmes d'emploi du temps se prêtant bien à cette hybridation, le modèle utilisant les variables p_{jk}, légèrement modifié, pourrait servir de générateur de colonnes dans une approche par génération de colonnes.

La décomposition de Benders logique, tout comme la génération de colonnes, s'articule autour d'un problème maître et de sous-problèmes résolus tour à tour. Le rôle du problème maître est de proposer une solution optimale au problème à résoudre tout en ignorant un certain nombre de contraintes difficiles à traiter. La réalisabilité de la solution proposée est ensuite évaluée par rapport à l'ensemble complet des contraintes spécifiées dans les sous-problèmes. Dans le cas du non-respect de certaines contraintes, des coupes linéaires s'ajoutent à la description du problème maître. Dans la version hybride, les sous-problèmes sont résolus en programmation par contraintes avec un modèle qui détermine des instanciations variable-valeur toujours incohérentes (ce qu'on appelle des *nogood*) qui, une fois transmises au problème maître, sont converties en coupes linéaires.

La recherche locale basée sur la PPC est issue du désir de résoudre des problèmes complexes à l'aide de méthodes heuristiques (chap. 9) tout en bénéficiant de la flexibilité de la programmation par contraintes et de ses puissants mécanismes de propagation. Le principe consiste à modéliser les opérateurs et leur voisinage comme un problème combinatoire que l'on résout à l'aide de la programmation par contraintes. Ainsi, puisque cette technique permet d'explorer un voisinage implicitement plutôt qu'explicitement, il est possible de définir des voisinages plus grands et il y a donc moins de risques de se faire piéger dans un minimum local.

RÉFÉRENCES BIBLIOGRAPHIQUES

[1] Apt, K., *Principles of Constraint Programming*, Cambridge University Press, 2003.

[2] Dechter, R., *Constraint Processing*, Morgan Kaufmann Publishers, 2003.

[3] Marriott, K. et P. Stuckey, *Programming with Constraints: An Introduction*, Boston, MIT Press, 1998.

[4] Régin, J.-C., «Generalized Arc Consistency for Global Cardinality Constraints», dans *Proc. of AAAI-96*, p. 209–215, AAAI Press/MIT Press, 1996.

[5] Baptiste, P., C. Le Pape et W. Nuijten, *Constraint-Based Scheduling*, Kluwer Academic Publishers, 2001.

[6] Pesant, G., «A Regular Language Membership Constraint for Finite Sequences of Variables», dans *Proc. of CP'04*, p. 482–495, Springer-Verlag LNCS 3258, 2004.

[7] Milano, M. (éd.), *Constraint and Integer Programming: Toward a Unified Methodology*, t. 27 de *Operations Research/Computer Science Interfaces*, Kluwer, 2004.