Chapitre 2

Programmation par contraintes

Nous présentons brièvement les principes de la programmation par contraintes pour la résolution de problèmes de satisfaction de contraintes et d'optimisation sous contraintes. Nous insisterons sur le rôle crucial joué par les stratégies de recherche notamment par le biais des heuristiques de sélection de variable et de valeur. Nous rappellerons aussi le principe des procédures d'optimisation qui transforme un problème d'optimisation sous contraintes en une série de problèmes de satisfaction de contraintes. Avant de conclure, nous passerons en revue les principaux solveurs rencontrés lors de nos recherches en précisant leurs fonctionnalités propres à l'ordonnancement.

Sommaire

La programmation par contraintes est une technique de résolution des problèmes combinatoires complexes issue de la programmation logique et de l'intelligence artificielle et apparue à la fin des années 1980. Elle consiste à modéliser un problème par un ensemble de relations logiques, des contraintes, imposant des conditions sur l'instanciation possible d'un ensemble de variables définissant une solution du problème. Un solveur de contraintes calcule une solution en instanciant chacune des variables à une valeur satisfaisant simultanément toutes les contraintes. De nos jours, de nombreuses techniques issues de la recherche opérationnelle, de la programmation mathématique ou même de la recherche locale sont appliquées grâce à la séparation entre un langage de modélisation déclaratif et les algorithmes employés durant la résolution. Les types d'application traités par la programmation par contraintes sont très variés. Parmi les domaines d'application les plus classiques, on pourra citer la gestion du temps ou d'affectation de

ressources, la planification et l'ordonnancement, mais aussi la simulation comportementale des systèmes, la vérification des spécifications ou le diagnostic des pannes. La mise en œuvre des contraintes dans un contexte industriel peut se faire sous diverses formes : certaines entreprises utilisent des solveurs commerciaux, d'autres font seulement appel occasionnellement à des bibliothèques de gestion de contraintes, d'autres enfin développent des prototypes servant uniquement à la modélisation du problème qui peuvent ensuite être recodés de manière plus efficace dans un langage de programmation traditionnel ou à l'aide d'un module numérique. En revanche, la manière de poser le problème peut modifier considérablement les performances du programme. De plus, l'évaluation du temps d'exécution peut être si subtile et dépendante du problème, qu'il n'est pas possible d'énoncer des règles claires. Ces aspects peuvent entraîner une frustration au regard des espérances qu'elle fait naître auprès des utilisateurs. Ainsi, il est critique d'évaluer la taille d'un problème et la combinatoire qu'il engendre avant d'espérer pouvoir le résoudre.

Les problèmes d'ordonnancement et de placement forment une classe de problèmes d'optimisation combinatoire généralement complexe. Cependant, la classification de ces problèmes et l'étude de leur complexité ont permis d'exhiber de nombreux problèmes polynomiaux. Un large éventail de méthodes complexes d'optimisation approchées ou exactes ont été appliquées sur les problèmes non polynomiaux avec des succès divers. La programmation par contraintes a connu quelques-uns de ses plus retentissants succès lors de la résolution de problèmes d'ordonnancement. En effet, la combinaison des méthodes de recherche opérationnelle (théorie des graphes, programmation mathématique, algorithmes de permutation), avec des modèles de représentation et de traitement des contraintes de l'intelligence artificielle (satisfaction de contraintes, propagation de contraintes, langages déclaratifs de modélisation) ont permis de mettre au point des outils facilitant l'interaction entre les modèles et les décideurs en ajoutant des méthodes d'analyse (vérification de la consistance, caractérisation de l'espace des solutions) à des algorithmes de résolution efficaces (recherche arborescente, recherche locale, optimisation). La conception de mécanismes efficaces et généraux de propagation de contraintes qui réduisent l'espace de recherche sera discutée au chapitre suivant dans le cas des contraintes temporelles et de partage de ressource utilisées dans les problèmes traités dans cette thèse.

La propagation des contraintes seule est généralement insuffisante pour exhiber une solution, il est alors nécessaire d'appliquer un algorithme de recherche pour trouver une ou plusieurs solutions. La méthode la plus commune est la recherche arborescente avec retour arrière, mais de nombreuses variantes existent incluant des algorithmes prospectifs, rétrospectifs ou mixtes. D'autres algorithmes de recherche s'inspirent de la recherche locale en adaptant les notions de voisinage ou de population. Il est donc nécessaire d'identifier les techniques adaptées à la résolution d'un type de problème donné. Les heuristiques de sélection déterminent l'ordre de traitement des variables et des valeurs dans une recherche arborescente et, de ce fait, influencent ses performances.

Nous discutons dans ce chapitre des principes fondateurs de la PPC. Le lecteur est invité à se référer à Rossi *et al.* [10] pour un état de l'art approfondi. Dans un premier temps, nous décrivons le modèle de déclaration des problèmes de satisfaction de contraintes et d'optimisation sous contraintes (section 2.1) avant de discuter des méthodes de résolution des problèmes à base de contraintes et des techniques permettant d'accélérer le processus de résolution (section 2.2). Ensuite, nous abordons quelques notions relatives aux stratégies de recherche (section 2.3) et à l'optimisation sous contraintes (section 2.4). Avant de conclure, nous présentons les principes de quelques solveurs de contraintes qui mettent à disposition des utilisateurs une bibliothèque pour la modélisation et la résolution de problèmes variés (section 2.5).

2.1 Modélisation d'un problème par des contraintes

Nous définissons dans cette section les différents éléments permettant de modéliser un problème combinatoire par une conjonction de contraintes. Dans une première partie, nous définissons les problèmes de satisfaction de contraintes puis nous présentons les problèmes d'optimisation sous contraintes dont les solutions maximisent ou minimisent la valeur d'une fonction de coût.

2.1.1 Problème de satisfaction de contraintes

Une contrainte est une relation logique établie entre différentes variables, chacune prenant sa valeur dans un ensemble qui lui est propre, appelé domaine. Une contrainte sur un ensemble de variables restreint les valeurs que peuvent prendre simultanément ses variables. Une contrainte est déclarative et relationnelle puisqu'elle définit une relation entre les variables sans spécifier de procédure opérationnelle pour assurer cette relation. L'arité d'une contrainte est égale à la cardinalité de son ensemble de variables, appelé son scope. Par exemple, la contrainte binaire $(x, y) \in \{(0, 1), (1, 2)\}$ précise que la variable x ne peut prendre la valeur 0 que si la variable y prend la valeur 1 et que la variable y peut prendre la valeur 2 uniquement si x prend la valeur 1. Une telle contrainte est définie en extension, c'est-à-dire en spécifiant explicitement les tuples de valeurs qu'elle autorise (ou interdit). Une contrainte peut également être définie en intension par une équation mathématique ($x+y \leq 4$), une relation logique simple ($x \leq 1 \Rightarrow y \neq 2$) ou même par un prédicat portant sur n variables ($\texttt{allDifferent}(x_1, \ldots, x_n)$). Une telle contrainte est appelée contrainte globale. Nous reviendrons sur la définition et l'intérêt des contraintes globales tout au long de cette thèse. La définition d'une nouvelle contrainte est aisée puisque la seule opération indispensable est le test de consistance qui détermine si une instanciation de ces variables satisfait la contrainte.

Un problème de satisfaction de contraintes (CSP) est défini formellement par un triplet $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ représentant respectivement un ensemble fini de variables $\mathcal{X}$, une fonction $\mathcal{D}$ affectant à chaque variable $x \in \mathcal{X}$ son domaine $\mathcal{D}(x)$ et un ensemble fini de contraintes $\mathcal{C}$. Le domaine initial d'une variable $\mathcal{D}_0(x)$ représente l'ensemble de valeurs auxquelles la variable x peut être instanciée avant le début de la résolution. Chaque contrainte $c \in \mathcal{C}$ est une relation multidirectionnelle portant sur un sous-ensemble de variables noté $var(c) \subseteq \mathcal{X}$ restreignant les valeurs que ces variables peuvent prendre simultanément. Les problèmes traités dans cette thèse sont des CSP discrets ($\mathcal{D}(x) \in \mathbb{Z}$) et statiques, c'est-à-dire l'ensemble des contraintes ne change pas au cours de la résolution.
Le graphe de contraintes d'un CSP binaire est un graphe simple où les sommets représentent les variables et les arêtes représentent les contraintes. Il y a une arête entre les sommets représentant les variables x et y si et seulement s'il existe une contrainte c telle que $var(c) = \{x, y\}$. Le concept de graphe de contraintes peut être défini pour des CSP quelconques (non binaires). Les contraintes ne sont alors pas représentées par des arêtes mais par des hyperarêtes qui relient des sous-ensembles de variables impliquées dans une même contrainte. Dans ce cas, on a un hypergraphe de contraintes. On appelle voisinage d'un sommet du graphe l'ensemble des sommets adjacents de ce graphe, c'est-à-dire la liste des sommets auxquels on peut accéder directement depuis le sommet courant (concept d'adjacence). Le degré d'un sommet du graphe est le nombre d'arêtes qui entrent et qui sortent du sommet en cours. Pour un graphe non orienté comme le graphe de contraintes, ceci correspond tout naturellement au cardinal de son voisinage. De même, on peut définir les graphes de consistance/inconsistance dont les sommets sont les valeurs et les hyperarcs sont les tuples autorisés/interdits.
Une instanciation $x \leftarrow v$ consiste à affecter à une variable x une valeur v appartenant à son domaine $\mathcal{D}(x)$. À chaque étape de la résolution, une affectation partielle $\mathcal{A}$ est définie comme l'ensemble des domaines courants de toutes les variables. Le domaine courant $\mathcal{D}(x)$ d'une variable x est toujours un sous-ensemble de son domaine initial $\mathcal{D}_0(x)$ (inclusion non stricte). On note $var(\mathcal{A})$ l'ensemble des variables instanciées, c'est-à-dire dont le domaine est réduit à un élément. Une affectation est dite partielle si $var(\mathcal{A}) \subsetneq \mathcal{X}$ ou totale si $var(\mathcal{A}) = \mathcal{X}$. Une affectation $\mathcal{A}'$ restreint une affectation partielle $\mathcal{A}$, noté $\mathcal{A}' \subseteq \mathcal{A}$, si le domaine de n'importe quelle variable x dans $\mathcal{A}'$ est un sous-ensemble de son domaine dans $\mathcal{A}$. Les valeurs minimum et maximum du domaine d'une variable entière x sont notées respectivement $\min(x)$ et $\max(x)$. Les notations $\min(x) \leftarrow v$ et $\max(x) \leftarrow v$ représentent respectivement les réductions du domaine à $\mathcal{D}(x) \cap [v, +\infty[$ et $\mathcal{D}(x) \cap]-\infty, v]$. La notation $x \not\leftarrow v$ indique la suppression de la valeur v du domaine $\mathcal{D}(x)$.
Une affectation viole une contrainte si toutes ses variables sont instanciées et que la relation associée à la contrainte n'est pas vérifiée. Une affectation est consistante si elle ne viole aucune contrainte et inconsistante dans le cas contraire. Une solution d'un CSP est donc une affectation totale consistante.
Résoudre un CSP consiste à exhiber une unique solution ou à montrer qu'aucune solution n'existe (le problème est irréalisable). *Prouver la consistance d'un CSP est un problème NP-complet mais exhiber une solution est un problème NP-difficile dans le cas général.* On peut également être intéressé par la détermination de plusieurs ou toutes les solutions d'un problème.

2.1.2 Problème d'optimisation sous contraintes

Dans de nombreux cas, des relations de préférence entre les solutions d'un CSP existent eu égard aux critères fixés par le décideur. L'optimisation consiste alors rechercher des solutions optimales d'un CSP au regard des relations de préférence du décideur. Nous nous intéresserons uniquement à des problèmes d'optimisation combinatoire monocritère, c'est-à-dire lorsque l'ensemble des solutions est discret et qu'il y a un critère d'optimalité unique. Ce critère d'optimalité est généralement associé à la maximisation/-minimisation d'une fonction objectif d'un sous-ensemble de variables vers les entiers.
Un problème d'optimisation sous contraintes (COP) est un CSP augmenté d'une fonction objectif f. Cette fonction est souvent modélisée par une variable dont le domaine est défini par les bornes supérieures et inférieures de f.

2.1.3 Exemple : le problème des n reines

Au cours de ce chapitre, nous illustrerons divers concepts sur le problème des n *reines*. Le but de ce problème, inspiré du jeu d'échec, est de placer n reines sur un échiquier de dimension $n \times n$ de manière à ce qu'aucune ne soit en prise. Deux reines sont en prises si elles sont sur la même ligne, la même colonne ou la même diagonale. Un échiquier classique comporte 8 lignes et 8 colonnes. Ce problème désormais classique est devenu une référence de mesure de performance des systèmes grâce à un énoncé simple masquant sa difficulté. Un modèle classique sans contraintes globales est défini dans la figure 2.1. En observant que deux reines ne peuvent pas être placées sur la même colonne, on peut imposer que la reine i soit sur la colonne i. Ainsi, la variable l_i de domaine $\{1, \ldots, n\}$ représente la ligne où est placée la reine dans la colonne i. Les contraintes (2.1) imposent que les reines soient sur des lignes différentes alors que les contraintes (2.2) et (2.3) imposent que deux reines soient placées sur des diagonales différentes.

$$l_i \neq l_j \qquad 1 \leq i < j \leq n \tag{2.1}$$
$$l_i \neq l_j + (j - i) \qquad 1 \leq i < j \leq n \tag{2.2}$$
$$l_i \neq l_j - (j - i) \qquad 1 \leq i < j \leq n \tag{2.3}$$

FIGURE 2.1 – Exemple de CSP : n reines.

La figure 2.2 montre plusieurs affectations partielles pour le problème des 4 reines dans lesquelles le placement d'une reine sur l'échiquier est représenté par un cercle dans la case correspondante. Les reines sont ordonnées de gauche à droite et les positions de haut en bas. On peut observer en figure 2.2 que l'affectation partielle $\{l_1 \leftarrow 1, l_2 \leftarrow 3, l_3 \leftarrow 2\}$ n'est pas consistante car elle viole une des contraintes (2.2). Au contraire, les affectations totales $\{l_1 \leftarrow 2, l_2 \leftarrow 4, l_3 \leftarrow 1, l_4 \leftarrow 3\}$ et $\{l_1 \leftarrow 3, l_2 \leftarrow 1, l_3 \leftarrow 4, l_4 \leftarrow 2\}$ sont les deux solutions symétriques des 4 reines.

Si l'on définit un COP à partir du problème des n reines en définissant la fonction objectif $min(l_1)$, alors le problème admet une unique solution optimale : $\{l_1 \leftarrow 2, l_2 \leftarrow 4, l_3 \leftarrow 1, l_4 \leftarrow 3\}$.

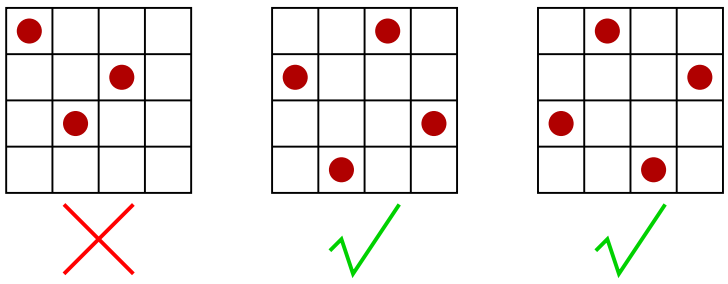


FIGURE 2.2 – Exemples d'affectation : 4 reines.

2.2 Méthodes de résolution

Les méthodes de résolution des CSPs sont génériques, c'est-à-dire qu'elles ne dépendent pas de l'instance à résoudre. Cependant, des techniques dédiées améliorent la résolution de différentes classes de problèmes. Dans le contexte d'un CSP statique et discret, nous discuterons de la réduction de l'espace de recherche par des techniques de consistance couplées si nécessaire à un algorithme de recherche arborescente.
Dans un premier temps, nous présentons deux algorithmes de recherche arborescente dont le comportement peut être amélioré par des techniques de consistance ou des contraintes globales. Ensuite, nous décrivons plusieurs algorithmes avancés, prospectifs ou rétrospectifs, dans le contexte d'une recherche complète par séparation et évaluation.

2.2.1 Algorithmes simples de recherche

L'algorithme *generate-and-test* énumère les affectations totales et vérifie leur consistance, c'est-à-dire qu'aucune contrainte n'est violée. En général, les affectations sont énumérées grâce à une recherche arborescente qui instancie itérativement les variables. Cet algorithme ne réveille les contraintes que lorsqu'une affectation est totale, mais considère par contre un nombre exponentiel d'affectations le rendant impraticable même pour des problèmes de petite taille. Des résultats de complexité ont montré que prouver la satisfiabilité d'un CSP était un problème NP-complet mais qu'exhiber une solution admissible ou optimale était un problème NP-difficile.
L'algorithme simple retour arrière (*backtrack*) [10] étend progressivement l'affectation vide en instanciant une nouvelle variable à chaque étape. L'algorithme vérifie alors que l'affectation partielle étendue est consistante avec les contraintes du problème. Dans le cas contraire, la dernière instanciation faite est remise en cause et l'on effectue une nouvelle instanciation. De cette manière, l'algorithme construit un arbre de recherche dont les nœuds représentent les affectations partielles testées. Cet algorithme teste l'ensemble des affectations possibles de manière implicite, c'est-à-dire sans les générer toutes. Le nombre d'affectations considéré est ainsi considérablement réduit, mais en revanche, la consistance des contraintes est vérifiée à chaque affectation partielle.

La figure 2.3 illustre la résolution du problème des 4 reines par l'algorithme *backtrack* qui place à chaque point de choix la première reine libre en partant de la gauche à la première position disponible en partant du haut. À chaque point de choix, l'affectation partielle courante est étendue en instanciant les variables et les valeurs selon l'ordre lexicographique. Dans ce cas précis, toutes les solutions symétriques sont éliminées en imposant que la première reine (à gauche) soit placée dans la partie haute de l'échiquier. Remarquez que la recherche continue après la découverte de la première solution puisqu'on cherche toutes les solutions à une symétrie près. L'algorithme *generate-and-test* consiste à déployer l'arbre entier.

La découverte redondante d'inconsistance locale due à la perte d'information sur l'inconsistance d'affectation partielle dégrade les performances de l'algorithme *backtrack*. Nous discuterons donc de l'utilisation active des contraintes pour supprimer les inconsistances, des algorithmes prospectifs qui anticipent les prochaines affectations pour réduire les domaines, et rétrospectifs qui utilisent les conflits pour remettre en cause les choix antérieurs.

2.2.2 Filtrage et propagation des contraintes

Le filtrage et la propagation des contraintes [10] permettent d'améliorer les capacités de résolution des CSP lors de la construction d'un arbre de recherche. Une fonction `revise()` associée à chaque contrainte réalise le filtrage des domaines, c'est-à-dire qu'elle supprime les valeurs inconsistantes des domaines de ces variables. Différents niveaux de consistance locale existent pour une même contrainte. La propagation calcule un point fixe global qui est atteint lorsque les techniques de consistance locale ne peuvent plus réaliser aucune inférence. En général, les algorithmes de consistance sont incomplets, c'est-à-dire qu'ils ne retirent pas toutes les valeurs inconsistantes des domaines.

Les opérations (2.4) et (2.5) définissent une fonction `revise()` pour la contrainte arithmétique $x \leq y$. Supposons que les domaines de x et y sont égaux à $\{1, 2, 3\}$, alors l'opération (2.4) réduit le domaine de x à $\{1, 2\}$ tandis que l'opération (2.5) réduit le domaine de y à $\{2, 3\}$. La fonction `revise()` a atteint

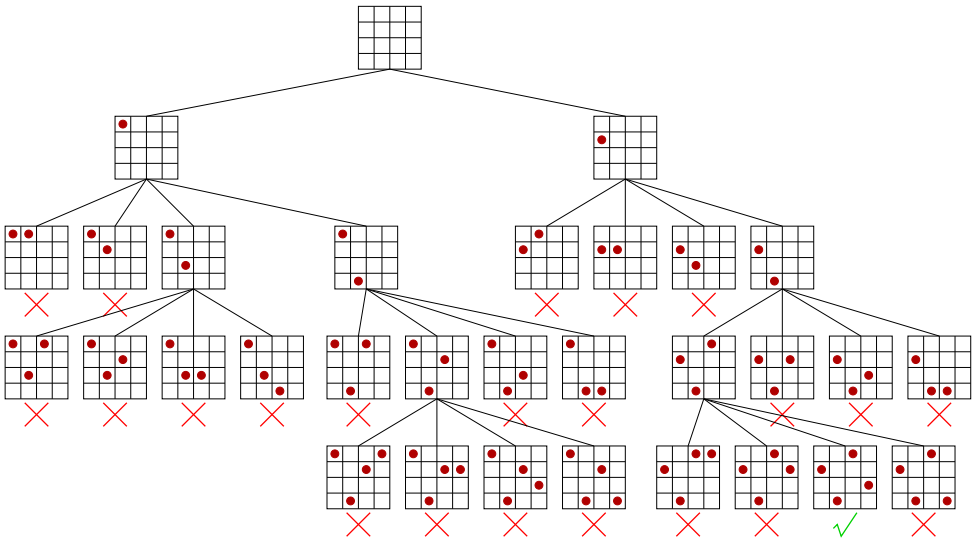


FIGURE 2.3 – Algorithme *backtrack* : 4 reines.

son point fixe puisqu'aucune règle ne peut plus produire d'inférence. Dans de nombreux cas, la fonction `revise()` doit être appelée plusieurs fois avant d'atteindre son point fixe.

$$\max(x) \leftarrow \max(y) - 1 \tag{2.4}$$
$$\min(y) \leftarrow \min(x) + 1 \tag{2.5}$$

Nous rappelons les principales techniques de consistance. Un CSP est dit consistant de nœud si pour toute variable x et pour toute valeur $v \in \mathcal{D}(x)$, l'affectation partielle $x \leftarrow v$ satisfait toutes les contraintes unaires, c'est-à-dire dans lesquelles x est l'unique variable non instanciée. Une contrainte est dite arc-consistante si pour chaque valeur de chaque variable x, il existe une affectation des autres variables telle que la contrainte soit satisfaite. Un CSP est dit arc-consistant lorsque toutes ses contraintes sont arc-consistantes. Néanmoins, il est insuffisant d'effectuer une révision unique par contrainte pour atteindre le point fixe global. Par conséquent, l'algorithme AC-1 révise toutes les contraintes jusqu'à ce qu'aucun domaine n'ait changé. L'algorithme AC-3 utilise une file de contraintes à réviser dans laquelle il ajoute les contraintes portant sur une variable dont le domaine a changé. En pratique, les algorithmes à partir d'AC-3 affinent le réveil des contraintes en fonction d'évènements sur les domaines définis par le solveur (réduction, suppression, instanciation ...). Toutefois, l'arc-consistance ne détecte pas les inconsistances dues à un ensemble de contraintes comme nous le verrons dans la section suivante.
Lorsque les domaines des variables sont trop grands, la mémoire requise pour stocker l'appartenance ou non de chaque valeur devient problématique. De la même manière, l'application des techniques de consistance pour chaque couple de variable et de valeur peut dégrader considérablement la vitesse de propagation par rapport à la réduction effective des domaines. On utilise alors la consistance de bornes qui consiste à raisonner sur la valeur minimale et maximale que les variables peuvent prendre. Pour certaines contraintes, la consistance de borne est très proche, voire égale à la consistance d'arc, notamment pour la contrainte $x \leq y$ discutée ci-dessus.

Ainsi, à chaque nœud de l'arbre de recherche, c'est-à-dire à chaque instanciation d'une variable, l'algorithme de recherche applique des techniques de consistance locale, propre à chaque contrainte, qui retirent des valeurs inconsistantes des domaines. À leur tour, les modifications des domaines peuvent inférer de nouvelles inconsistances. Ce mécanisme, appelé propagation, continue jusqu'à ce qu'un point fixe global soit atteint. L'algorithme fait alors un nouveau choix de variable et de valeur et teste la nouvelle instanciation. Si durant le filtrage d'une contrainte, le domaine d'une variable devient vide, alors l'instanciation courante est inconsistante. Le nœud est fermé et l'algorithme passe au nœud suivant s'il existe. Cette opération permet d'agir sur les domaines de toutes les variables d'un CSP durant la construction des affectations partielles réduisant ainsi le nombre de nœuds composant l'arbre.

2.2.3 Contraintes globales

La modélisation des problèmes complexes est facilitée par l'utilisation de contraintes hétérogènes agissant indépendamment sur de petits ensembles de variables. Toutefois, la détection locale des inconsistances affaiblit la réduction des domaines. Les contraintes globales corrigent partiellement ce comportement en utilisant l'information sémantique issue de raisonnements sur des sous-problèmes. Elles permettent généralement d'augmenter l'efficacité du filtrage ou de réduire les temps de calcul.

Par exemple, l'arc-consistance ne détecte pas l'inconsistance globale du CSP présenté dans la figure 2.4. En effet, l'inconsistance est issue des trois contraintes qui imposent que les trois variables prennent des

$$x \neq y \qquad x \neq z \qquad z \neq y$$
$$\mathcal{D}_0(x) = \mathcal{D}_0(y) = \mathcal{D}_0(z) = \{0, 1\}$$

FIGURE 2.4 – Exemple de CSP où l'arc-consistance est incomplète.

valeurs distinctes alors que l'union de leurs domaines ne contient que deux valeurs. La contrainte globale $\texttt{allDifferent}\,(x_1, \ldots, x_n)$ qui impose que ses variables prennent des valeurs distinctes détecte cette inconsistance triviale. Ses algorithmes de filtrage reposent sur le calcul de couplages maximaux dans un graphe biparti dont les deux ensembles de sommets représentent respectivement les variables et les valeurs alors que les arêtes représentent les instanciations possibles [11].

Un second modèle classique pour le problème des reines défini dans la figure 2.5 utilise des contraintes globales `allDifferent` pour représenter les cliques d'inégalités binaires. On introduit généralement les variables auxiliaires x_i et y_i par le biais des contraintes de liaison (2.6). Les variables auxiliaires correspondent aux projections sur la première colonne de la reine i en suivant les diagonales. La contrainte (2.7) impose que les reines soient sur des colonnes différentes alors que les contraintes (2.8) et (2.9) imposent que deux reines soient placées sur des diagonales différentes.

$$x_i = l_i - i \qquad y_i = l_i + i \qquad 1 \leq i \leq n \tag{2.6}$$
$$\texttt{allDifferent}\,(l_1, \ldots, l_n) \tag{2.7}$$
$$\texttt{allDifferent}\,(x_1, \ldots, x_n) \tag{2.8}$$
$$\texttt{allDifferent}\,(y_1, \ldots, y_n) \tag{2.9}$$

FIGURE 2.5 – Exemple de CSP utilisant des contraintes globales : n reines.

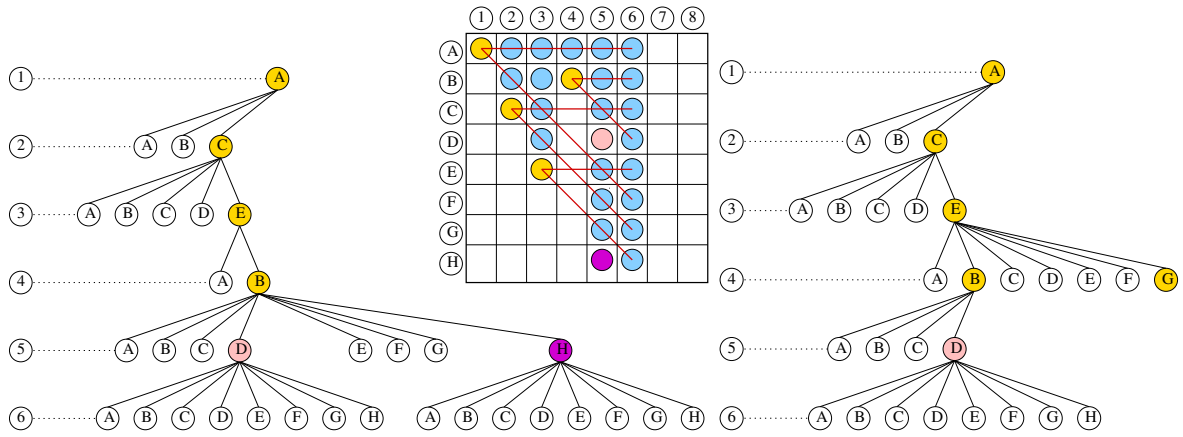
Beldiceanu et Demassey [12] proposent une classification complète des contraintes globales accompagnée d'une description de leurs algorithmes de filtrage dans laquelle figurent la plupart des contraintes discutées dans cette thèse.

2.2.4 Algorithmes avancés de recherche

Une technique prospective simple pour anticiper les effets d'une instanciation est nommée *forward checking*. On vérifie que les variables non instanciées peuvent chacune prendre une valeur consistante lorsque l'affectation partielle courante est étendue par l'instanciation d'une nouvelle variable. Les techniques de *look-ahead* sont plus lentes, mais procurent un meilleur filtrage basé sur l'arc-consistance. L'arc-consistance est appliquée sur toutes les variables non instanciées pour chaque affectation étendue. Ainsi, les techniques de *forward checking* considèrent une seule variable non instanciée à la fois pour la consistance alors que celles de *look-ahead* considèrent une paire de variables non instanciées.

D'un autre côté, les techniques rétrospectives remettent en cause l'affectation partielle lorsqu'une inconsistance est détectée lors de la propagation. Le *backtrack chronologique* consiste simplement à remettre en cause le dernier choix. Cet algorithme est sensible au phénomène de *thrashing* où des inconsistances

dues à un nombre restreint d'instanciations sont redécouvertes de manière redondante lors de l'exploration de l'arbre de recherche. La figure 2.6(a) illustre ce phénomène sur le problème des 8 reines lors de la remise en cause du placement de la reine 5 (colonnes D et H) car les conflits sur le placement de la reine 6 sont indépendants de cette décision.



(a) Thrashing : backtrack chronologique. (b) Conflict directed backjumping.

FIGURE 2.6 – Algorithmes de recherche rétrospectifs : 8 reines.

Le *conflit directed backjumping* ou *intelligent backtracking* [13] consiste à calculer une explication à un backtrack puis à remettre en cause le choix le plus récent de cette explication. Chaque échec sur le choix d'une valeur est expliqué par les précédents choix qui entrent en conflit. Si toutes les valeurs d'un domaine ont été testées sans succès, l'explication de cet échec est l'union des explications des valeurs du domaine. Par conséquent, le placement de la reine 4 est remis en cause dans la figure 2.6(b) lors du backtrack après la tentative de placement de la reine 6 car le conflit est expliqué par les choix portant sur les reines 1, 2, 3 et 4.
Le *dynamic backtracking* [14] utilise un mécanisme similaire mais sans remettre en cause les choix indépendants de l'explication. Par exemple, la prochaine décision du *dynamic backtracking* au nœud G de la reine 4 dans la figure 2.6(b) est le placement de la reine 5 dans la colonne D.
Le *nogood recording* [15] consiste à mémoriser les causes des conflits puis à utiliser des techniques de propagation inspirées des problèmes de satisfiabilité booléenne (SAT).

2.3 Stratégies de recherche

Les techniques de consistance étant incomplètes, les algorithmes de recherche résolvent les disjonctions restantes par des méthodes de séparation. Le nombre de nœuds et le temps de résolution d'un modèle peuvent varier considérablement en fonction de la méthode de séparation et des heuristiques de sélection employées.

2.3.1 Méthodes de séparation

À chaque création d'un nœud de l'arbre de recherche, l'algorithme crée un point de choix, c'est-à-dire qu'il considère successivement la résolution de sous-problèmes disjoints dans chaque branche. Dans un point de choix binaire, on peut considérer l'ajout d'une contrainte C dans la première branche et $\neg C$ dans la seconde grâce à la réversibilité des contraintes. Par exemple, la technique de *variable ordering* impose une relation d'ordre sur une paire de variables x et y : $x < y \vee x \geq y$. Cependant, on peut se limiter à considérer des points de choix opérant des restrictions sur le domaine d'une unique variable non instanciée. En effet, le mécanisme de réification d'une contrainte, c'est-à-dire l'association d'une variable booléenne indiquant la satisfaction de la contrainte, permet la transformation d'un point de choix sur des contraintes en un ou plusieurs autres portant sur des variables. De plus, la plupart des algorithmes

rétrospectifs ou techniques d'apprentissage utilisent des variables car leur domaine est restauré lors d'un retour arrière alors que les contraintes sont généralement supprimées. Les techniques standards opèrent généralement une restriction sur le domaine d'une seule variable $x \in \mathcal{X}$ choisie par une heuristique de sélection de variable.
Par exemple, un point de choix appliquant le *standard labeling* considère l'affectation d'une valeur $v \in \mathcal{D}(x)$ à la variable ou bien sa suppression : $x = v \vee x \neq v$. De la même manière, un point de choix appliquant le *domain splitting* restreint la variable x à des valeurs inférieures ou supérieures à une valeur seuil $v \in \mathcal{D}(x) : x < v \vee x \geq v$. La valeur v est choisie par une heuristique de sélection de valeur. On peut définir des branchements n-aires à partir des précédents en testant une valeur différente dans chaque branche (*standard labeling*) et en restreignant les valeurs de x à des intervalles disjoints dans chaque branche (*domain splitting*).
On appelle variables de décision, un sous-ensemble de variables dont l'instanciation provoque une affectation totale par propagation. Les méthodes de séparation peuvent limiter la profondeur de l'arbre en se restreignant à un sous-ensemble de variables de décision. Par exemple, les variables l_i du modèle des n reines en figure 2.5 sont des variables de décision. Cependant, on peut leur substituer les variables auxiliaires x_i et y_i qui sont aussi des variables de décision. Par défaut, toutes les variables du problème sont des variables de décision.
La programmation par contraintes offre donc un cadre flexible pour la définition de méthodes de séparation de par la structure des points de choix et l'utilisation d'heuristiques de sélection évoquée ci-dessous.

2.3.2 Heuristiques de sélection

Les heuristiques de sélection sont des règles qui déterminent (a) l'ordre suivant lequel on va choisir les variables pour les instancier, ainsi que (b) l'ordre suivant lequel on va choisir les valeurs pour les variables. Une bonne heuristique de sélection peut avoir un impact important sur la résolution d'un problème de décision et accélérer l'obtention de solutions (la détection d'échecs en cas de problèmes insolubles) pendant l'exploration de l'espace de recherche. Les algorithmes de résolution qui utilisent des heuristiques de sélection assurent, dans la plupart des cas, une résolution plus rapide que celle obtenue par des algorithmes sans heuristique de sélection. Une heuristique est statique si l'ordre est préalablement fixé ou dynamique s'il évolue au cours de la résolution. Les heuristiques de sélection sur des variables ensemblistes ne seront pas abordées dans le cadre de cette thèse.

2.3.2.1 Sélection de variable

Les heuristiques de sélection de variable ont souvent une influence critique sur la forme et la taille de l'arbre de recherche. Cette influence est toutefois plus restreinte lorsqu'on cherche toutes les solutions d'un CSP. De nombreuses heuristiques observent le principe *first-fail* énoncé par Haralick et Elliott [16] : « To succeed, try first where you are most likely to fail » (Pour réussir, essaie d'abord là où il est le plus probable d'échouer). Les heuristiques dynamiques exploitent souvent les informations sur les domaines et degrés des variables. On retrouve entre autres les heuristiques suivantes.

lex ordonne les variables selon l'ordre lexicographique (statique).

random sélectionne aléatoirement une variable non instanciée.

min-domain ou dom sélectionne la variable non instanciée qui a le plus petit domaine de valeurs.

degree ou deg l'heuristique du degré [17] ordonne les variables selon leurs degrés décroissants.

dynamic-degree ou ddeg l'heuristique du degré dynamique sélectionne la variable qui a le degré dynamique le plus élevé. Le degré dynamique est calculé pour l'affectation partielle courante.

weighted-degree ou wdeg l'heuristique du degré pondéré [18] se base sur l'apprentissage du poids de chaque contrainte en exploitant les étapes précédentes de la résolution. Le poids d'une contrainte est égal au nombre d'échecs provoqués par celle-ci. Elle sélectionne la variable dont la somme des degrés dynamiques pondérés est maximale.

dom/deg, dom/ddeg et dom/wdeg il s'agit de combinaisons des heuristiques précédentes dans laquelle on sélectionne la variable dont le quotient de la taille du domaine sur le degré est minimal.

impact l'impact d'une variable ou d'une affectation mesure son influence sur la réduction de l'espace de recherche. Leur apprentissage se base sur la surveillance des réductions de domaine pendant

la recherche. La recherche basée sur les impacts [19] sélectionne la variable d'impact maximal et l'instancie à sa valeur d'impact minimal.

min-conflict contrairement aux heuristiques présentées ci-dessus, l'heuristique du conflit minimum ordonne les valeurs. Il s'agit de choisir une valeur qui permettra à l'instanciation partielle courante d'être étendue à une solution. Une mesure commune pour juger vraisemblable la participation d'une valeur à une solution est de considérer le nombre de valeurs qui ne sont pas en conflit avec la valeur en question. Ainsi, la valeur qui est compatible avec la plupart des valeurs des variables non instanciées est essayée en premier.

Des techniques de redémarrage peuvent souvent être avantageusement combinées aux heuristiques par apprentissage (*wdeg*, *dom/wdeg*, *impact*) ou intégrant un élément de randomisation. Même si les différentes heuristiques de sélection sont en général incomparables entre elles, dans de nombreux cas, notamment sur des problèmes structurés, *dom/wdeg* fournit de très bons résultats.

2.3.2.2 Sélection de valeur

Une heuristique de sélection de valeur détermine la prochaine valeur du domaine à tester. Elle définit donc l'ordre d'exploration des branches d'un point de choix et dépend généralement du problème. Imaginons l'existence d'un oracle fournissant une valeur consistante pour chaque point de choix sur une variable, alors l'algorithme de recherche trouvera la première solution sans retour arrière quel que soit l'ordre d'énumération des variables. Ces heuristiques observent souvent le principe *succeed-first* qui consiste à choisir la valeur ayant la plus grande probabilité d'appartenir à une solution. L'heuristique **minVal** sélectionne la plus petite valeur du domaine, **maxVal** sélectionne la plus grande, alors que **randVal** sélectionne aléatoirement une valeur. Lorsque les domaines des variables sont énumérés, on peut appliquer l'heuristique **midVal** qui sélectionne la valeur médiane du domaine.

2.3.2.3 Exemple et discussion

Nous montrons sur un exemple l'influence des heuristiques de sélection sur la résolution du problème des 4 reines par l'algorithme *backtrack* avec un branchement n-aire par *standard labeling*. La figure 2.3 en page 14 représente l'arbre de recherche obtenu avec les heuristiques de sélection *lex* et *minVal*. La figure 2.7 représente l'arbre obtenu avec une heuristique de sélection de variable imposant un ordre statique $\{l_1, l_2, l_4, l_3\}$ combinée à une heuristique de sélection de valeur *maxVal*. L'avantage relatif à la découverte plus précoce de la solution est contrebalancé par l'augmentation de la taille de l'arbre établissant son unicité.

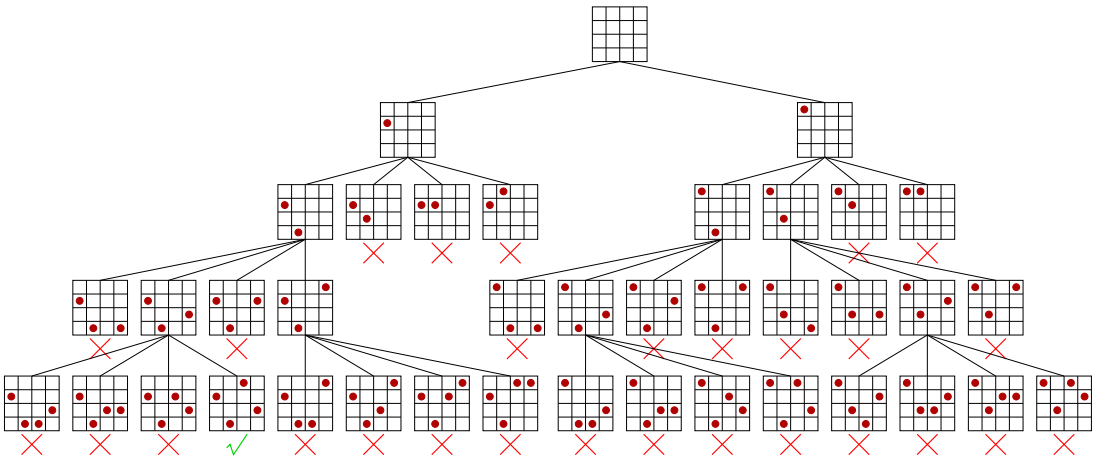


FIGURE 2.7 – Influence des heuristiques de sélection sur l'algorithme *backtrack* : 4 reines.

Il est important de remarquer que les heuristiques disposent de moins d'information au début de la recherche alors que les premiers points de choix ont une importance cruciale. Cette situation peut mener aux situations de *thrashing* décrites précédemment puisque les heuristiques ne sont pas infaillibles.

Cependant, l'exploration ne devrait pas trop s'éloigner des choix de l'heuristique lorsque celle-ci est très bien adaptée. La recherche en profondeur, utilisée par exemple dans l'algorithme de simple retour arrière, s'éloigne rapidement des choix initiaux de l'heuristique. Pour pallier ce problème, la *limited discrepancies search* (LDS) explore l'arbre de manière à remettre en cause un nombre minimum des choix initiaux de l'heuristique [20]. La figure 2.8 compare l'ordre d'exploration des branches d'une recherche en profondeur (à gauche) à LDS (au milieu) et une de ses variantes *depth-bounded discrepancies search* (DBDS). Les

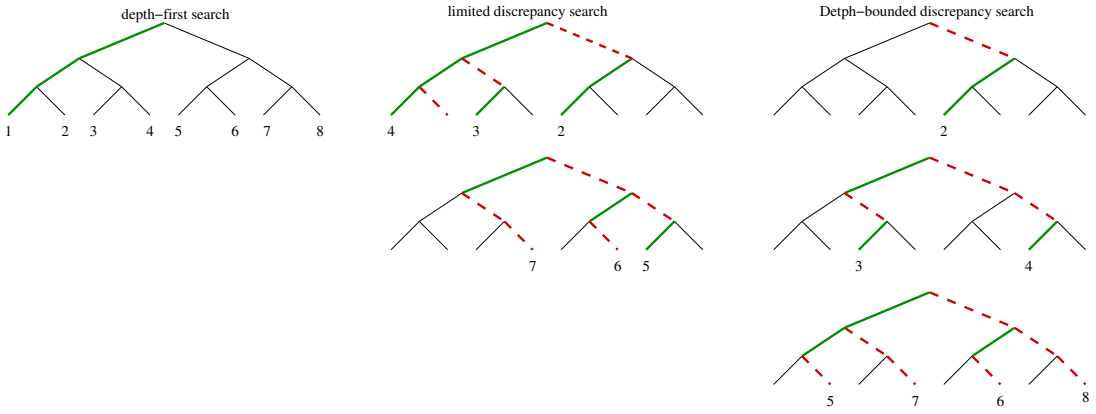


FIGURE 2.8 – Ordre d'exploration des branches par différentes recherches arborescentes.

branches en gras indiquent le respect des choix initiaux et celles en pointillés indiquent leur remise en cause. LDS et DBDS remettent en cause prioritairement les choix les plus précoces considérés moins fiables. Au contraire de LDS, DBDS cherche d'abord à minimiser la profondeur du dernier choix remis en cause plutôt que leur nombre.

2.4 Procédures d'optimisation

En pratique, résoudre un problème d'optimisation sous contraintes (COP) consiste à trouver une ou les solutions du CSP dont la valeur de la variable objectif est minimale ou maximale. Les extensions dédiées à l'optimisation reposent sur la résolution d'une série de problèmes de satisfaction de contraintes. La répartition et le nombre (au pire cas) de sous-problèmes satisfiables et insatisfiables caractérisent ces procédures. L'évaluation initiale de l'objectif a une influence majeure sur la construction de la série qui dépend également de l'algorithme de résolution des sous-problèmes. Le choix d'une procédure d'optimisation repose souvent sur l'analyse du modèle et des fonctionnalités du solveur de contraintes. La combinaison judicieuse d'une procédure et d'un algorithme de résolution des sous-problèmes est un élément clé d'une approche en optimisation sous contraintes, notamment pour les problèmes traités dans cette thèse.
Dans cette section, nous présentons d'abord deux procédures classiques bottom-up et top-down respectivement orientées vers l'insatisfiabilité et la satisfiabilité des sous-problèmes. Ensuite, nous décrivons plusieurs variantes complètes et incomplètes qui seront discutées ou utilisées ultérieurement dans ce manuscrit. Nous avons jugé intéressant de regrouper les descriptions de ces procédures, mais il est possible de les lire au cas par cas. Pour chaque procédure, un tableau récapitulera le nombre de sous-problèmes satisfiables et insatisfiables en fonction de la satisfiabilité du COP.

Sans perte de généralité, nous supposons que nous minimisons une fonction objectif représentée par une variable entière obj dont le domaine initial $\mathcal{D}_0(obj) = [lb, ub[$ est un intervalle fini non vide. Si problème est satisfiable, la valeur de l'optimum est notée opt. Une procédure d'optimisation réduit itérativement le domaine de l'objectif jusqu'à atteindre une condition d'arrêt : $lb = ub$. Lorsque le problème est satisfiable, la nouvelle valeur ub appartient à $\mathcal{D}_0$. La fonction `solve`(x, y) applique un algorithme de recherche complet pour résoudre le CSP où le domaine de l'objectif est réduit à l'intervalle $[x, y[$. La résolution s'achève à la découverte d'une solution et la fonction `solve`(x, y) renvoie la valeur de la fonction objectif. Lorsque le sous-problème est insatisfiable ou que la résolution est interrompue, la fonction renvoie la

valeur null.

À titre d'exemple, nous considérerons un COP où $\mathcal{D}_0(obj) = [0, 16[$ et dont les coûts des solutions appartiennent à l'ensemble $\{9, 11, 13, 15\}$. Dans les figures 2.9, 2.10, 2.11 et 2.12, le domaine initial est représenté par un tableau où les cellules contenant des valeurs correspondant à une solution sont indiquées par une étoile. La résolution d'un sous-problème par la fonction `solve(`x, y`)` est représentée par une flèche dont le numéro correspond à la position du sous-problème dans la série. Lorsque le sous-problème est insatisfiable, la flèche part du coin supérieur gauche de la cellule contenant l'ancienne valeur lb et pointe vers le coin supérieur gauche de la cellule contenant la nouvelle valeur lb (orientation gauche – droite). Lorsque le sous-problème est satisfiable, la flèche part du milieu de la cellule contenant l'ancienne valeur ub et pointe vers le milieu de la cellule contenant la nouvelle valeur ub (orientation droite – gauche).

2.4.1 Procédure bottom-up

La procédure bottom-up incrémente la borne inférieure lb de la fonction objectif lorsque le problème de satisfaction de contraintes `solve(`$lb, lb + 1$`)` est insatisfiable jusqu'à ce qu'une solution optimale soit trouvée. Lorsque le COP est satisfiable, la procédure résout $opt - lb$ problèmes insatisfiables avant de résoudre un unique problème satisfiable fournissant une solution optimale. Le COP est implicitement considéré comme étant satisfiable en regard de l'efficacité amoindrie de la procédure dans le cas contraire. En effet, lorsque le COP n'est pas satisfiable, il est plus efficace de résoudre directement le problème `solve(`lb, ub`)` plutôt que $ub - lb$ sous-problèmes disjoints. La construction des arbres de recherche pour des problèmes voisins peut dégrader les performances à cause de la propagation redondante et de la nécessité de recréer les points de choix. En fait, L'efficacité de bottom-up dépend de la qualité de la borne inférieure initiale et de la capacité de l'algorithme de résolution à prouver l'insatisfiabilité des sous-problèmes, généralement grâce à des techniques avancées de propagation et de filtrage. Il peut être intéressant d'utiliser une borne inférieure dédiée fournissant une garantie, de relever le niveau de consistance ou de renforcer la propagation par des techniques prospectives pour réduire la taille des arbres de recherche et par conséquent les redondances. Par contre, l'algorithme dispose d'informations plus précises car la valeur de l'objectif est fortement contrainte.
Nous verrons que le nombre de sous-problèmes traités par bottom-up sur l'exemple de la figure 2.9 est plus élevé que celui des autres procédures. En pratique, la difficulté moindre des premières étapes peut compenser le nombre plus élevé de sous-problèmes.

Procédure bottom-up(lb,ub)

tant que $lb < ub$ **faire**
  si `solve(`lb,$lb+1$`)` = null **alors** $lb \leftarrow lb + 1$;
  sinon $ub \leftarrow lb$;

	#sous-problèmes	
COP	$\#SAT$	$\#UNSAT$
SAT	1	$opt - lb$
UNSAT	0	$ub - lb$

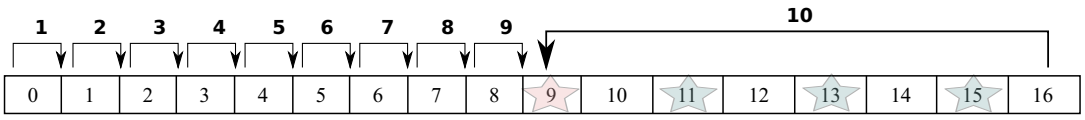


FIGURE 2.9 – Description de la procédure d'optimisation bottom-up.

2.4.2 Procédure top-down

Le succès des métaheuristiques s'explique entre autres par la découverte rapide de très bonnes solutions sans se préoccuper d'atteindre la preuve d'optimalité. Un objectif monocritère représente rarement la complexité d'une situation réelle et le bon sens recommande d'examiner des solutions diverses dont la qualité est jugée satisfaisante. Pour satisfaire cette demande, la stratégie top-down résout une série de problèmes de satisfaction dont les solutions améliorent la borne supérieure courante jusqu'à ce que le sous-problème devienne insatisfiable. Elle permet ainsi la découverte rapide de « bonnes » solutions alors que bottom-up fournit une unique solution optimale à la fin du calcul. La procédure débute par la recherche d'une première solution réalisable du COP par l'application de `solve(`lb,ub`)`. Lorsqu'une solution existe,

une boucle construit et résout des sous-problèmes de satisfaction dont les solutions améliorent la borne supérieure courante. Pour ce faire, le solveur doit proposer un service de coupe par le biais de la fonction `postDynamicCut`. Une coupe est une contrainte dont l'ajout est permanent contrairement à une contrainte qui est supprimée lors d'un retour arrière. Après la découverte d'une solution, une coupe restreint le domaine de l'objectif à des valeurs inférieures à la nouvelle borne supérieure. La recherche arborescente est ensuite reprise, grâce à la méthode `resumeSolve()`, à partir du dernier point de choix réalisable après l'ajout de la coupe. Contrairement à bottom-up, la structure des sous-problèmes est exploitée pour construire un unique arbre de recherche, au prix de services supplémentaires (interruption – reprise de la recherche, ajout de coupes). La procédure top-down résout un unique sous-problème lorsque le COP est insatisfiable. Dans le cas contraire, le nombre de sous-problèmes dépend des solutions améliorantes découvertes. La borne supérieure influence la réduction des domaines et par conséquent les informations utiles à l'algorithme de recherche. Ainsi, le nombre maximal de sous-problèmes $ub - opt$. Au contraire, la borne inférieure de l'objectif a une influence marginale sur la procédure mais peut éventuellement faciliter la preuve d'optimalité, c'est-à-dire la résolution du dernier sous-problème insatisfiable.
On peut constater sur l'exemple de la figure 2.10 que top-down ne visite pas toutes les solutions avant de découvrir l'optimum. En effet, la résolution du troisième sous-problème fournit la solution optimale avant la solution améliorante. Ce comportement illustre l'influence des heuristiques de sélection qui guident quelquefois la recherche vers de bonnes solutions même lorsque l'objectif est faiblement contraint.

Procédure top-down(lb,ub)

```
sol ← solve(lb,ub) ;
tant que sol ≠ null faire
    postDynamicCut(ub ← sol);
    sol ← resumeSolve();
lb ← ub;
```

	#sous-problèmes	
COP	$\#SAT$	$\#UNSAT$
SAT	$\leq ub - opt$	$\min(1, opt - lb)$
UNSAT	0	1

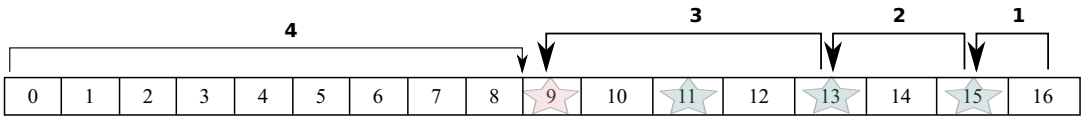


FIGURE 2.10 – Description de la procédure d'optimisation top-down.

2.4.3 Procédure dichotomic-bottom-up

La procédure dichotomic-bottom-up améliore bottom-up grâce à un partitionnement dichotomique du domaine de l'objectif. Elle considère ainsi un nombre restreint de sous-problèmes dont certains fournissent des solutions intermédiaires. À chaque itération, on résout un sous-problème dans lequel la valeur de l'objectif doit appartenir à la première moitié de son domaine courant. On actualise la borne supérieure de l'objectif lorsque le sous-problème fournit une solution et sa borne inférieure dans le cas contraire. Lorsque le COP est insatisfiable, le nombre de sous-problèmes $O(\log_2(ub - lb + 1))$ est considérablement réduit par rapport à bottom-up. Lorsque le COP est satisfiable, la procédure traite au plus $\log_2(ub - opt)$ sous-problèmes insatisfiables et $\log_2(opt - lb) + 1$ sous-problèmes satisfiables. Le nombre de sous-problèmes peut alors dépasser celui de bottom-up. Par exemple, la découverte des solutions intermédiaires freine la résolution si la borne inférieure initiale est égale à l'optimum.
L'exemple de la figure 2.11 illustre le mouvement de va-et-vient lors de la réduction du domaine de l'objectif par dichotomic-bottom-up qui fournit une solution intermédiaire dès le deuxième sous-problème. La découverte de la solution intermédiaire illustre la sensibilité des heuristiques de sélection au domaine de l'objectif, car le deuxième sous-problème de top-down fournit directement la solution optimale.

2.4.4 Procédures incomplètes

L'évaluation initiale de l'objectif joue donc un rôle crucial concernant le nombre de sous-problèmes mais aussi leur résolution. Des méthodes d'approximation peuvent éventuellement améliorer ou garan-

Procédure dichotomic-bottom-up(lb,ub)

tant que $lb < ub$ **faire**
  $sol \leftarrow$ `solve` $(lb, \lceil \frac{lb+ub}{2} \rceil)$;
  si $sol =$ `null` **alors** $lb \leftarrow \lceil \frac{lb+ub}{2} \rceil$;
  sinon $ub \leftarrow sol$;

	#sous-problèmes	
COP	$\#SAT$	$\#UNSAT$
SAT	$\leq \log_2(ub - opt)$	$\leq \log_2(opt - lb + 1)$
UNSAT	0	$\log_2(ub - lb) + 1$

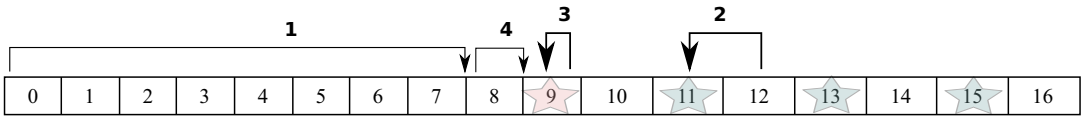


FIGURE 2.11 – Description de la procédure d'optimisation dichotomic-bottom-up.

tir la qualité de cette évaluation initiale. Cependant, elles demandent un effort de compréhension et d'implémentation voire d'adaptation en présence de contraintes additionnelles. Nous présentons plusieurs procédures d'optimisation incomplètes qui affinent l'évaluation initiale de l'objectif.

2.4.4.1 Procédure destructive-lower-bound

La procédure destructive-lower-bound inspirée de bottom-up affine l'évaluation de la borne inférieure sans appliquer de recherche arborescente. Leur principe consiste à fixer une valeur maximale de l'objectif,

Procédure destructive-lower-bound(lb,ub)

tant que $(lb < ub) \wedge$ `isNotFeasible(`lb,$lb+1$`)` **faire**
  $lb \leftarrow lb + 1$;

puis essayer de contredire (détruire) la satisfiabilité du problème ainsi reduit. La fonction `isNotFeasible` applique un test de consistance basé sur la propagation initiale éventuellement complété par différentes techniques prospectives comme du *shaving*. La borne inférieure lb est incrémentée tant que le sous-problème `isNotFeasible` $(lb, lb + 1)$ est prouvé insatisfiable sans retour arrière. Dans le cas contraire, lb est une borne inférieure valide pour le problème. Elle est souvent utilisée pour comparer l'efficacité de différents filtrages car son résultat ne dépend ni de l'algorithme de recherche ni des heuristiques de sélection. Elle permet aussi d'affiner rapidement l'évaluation de la qualité des solutions fournies par la procédure top-down lorsque la preuve d'optimalité n'est pas atteinte, par exemple lors d'une interruption de la recherche (limites de temps, nœuds ...). Par contre, elle est inutile dans le contexte de procédures inspirées de bottom-up qui fournissent une meilleure évaluation de la borne inférieure puisque les preuves sont obtenues par des recherches arborescentes complètes.

2.4.4.2 Procédure destructive-upper-bound

La procédure destructive-upper-bound offre une garantie sur la nouvelle évaluation des bornes inférieure et supérieure de l'objectif. Elle consiste à partitionner le domaine de l'objectif en intervalles $I_k = \big[lb + (2^k - 1), lb + (2^{k+1} - 1)\big[$. Lorsque le problème d'optimisation est insatisfiable, la procédure résout un nombre de sous-problèmes identique à celui de dichotomic-bottom-up mais le partitionnement du domaine de l'objectif est différent. Dans le cas contraire, elle résout $\log_2(opt - lb + 1)$ sous-problèmes insatisfiables améliorant l'évaluation de la borne inférieure et un unique sous-problème satisfiable garantissant que la solution appartient au plus petit intervalle I_k contenant une solution. Les conditions d'arrêt sont différentes puisque la procédure se contente d'affiner l'évaluation de l'objectif et ne garantit pas l'optimalité de la solution. Lorsque le problème est satisfiable, la nouvelle évaluation ub respecte la relation $ub \geq lb$ et, dans le cas contraire, la valeur ub n'est pas modifiée.
La figure 2.12 illustre la résolution du problème en appliquant successivement destructive-upper-bound puis dichotomic-bottom-up (flèches en pointillés). Dans notre exemple, il est préférable de ne pas appliquer la procédure destructive-upper-bound avant la procédure dichotomic-bottom-up. En effet, les

procédures dichotomic-bottom-up (figure 2.11) et destructive-upper-bound (figure 2.11) accomplissent le même nombre d'étapes pour (a) résoudre le problème d'optimisation et (b) améliorer l'évaluation de l'objectif. Par contre, cette combinaison est efficace lorsque la borne supérieure initiale est mauvaise. Par exemple, supposons que l'optimum est égal à 1, la combinaison traiterait deux sous-problèmes alors que dichotomic-bottom-up traiterait au pire cinq sous-problèmes. En effet, la combinaison supprime l'influence de la borne supérieure initiale sur le nombre de sous-problèmes, car la construction des intervalles I_k fournit une garantie, certes faible, sur la nouvelle borne supérieure ($log_2(ub-lb) \leq \log_2(opt-lb+1)+1$). Ainsi, le nombre de sous-problèmes dépend seulement de la différence entre l'optimum et la borne inférieure et non de la différence entre la borne supérieure et inférieure.

Procédure destructive-upper-bound(lb,ub)

$k \leftarrow 0$;
tant que $lb + (2^k - 1) < ub$ **faire**
  $sol \leftarrow$ `solve`$(lb + (2^k - 1), lb + (2^{k+1} - 1))$;
  si $sol =$ `null` **alors** $k \leftarrow k + 1$;
  sinon $ub \leftarrow sol$;**break** ;
$lb \leftarrow lb + (2^k - 1)$;

	#sous-problèmes	
COP	$\#SAT$	$\#UNSAT$
SAT	1	$\log_2(opt - lb + 1)$
UNSAT	0	$\log_2(ub - lb) + 1$

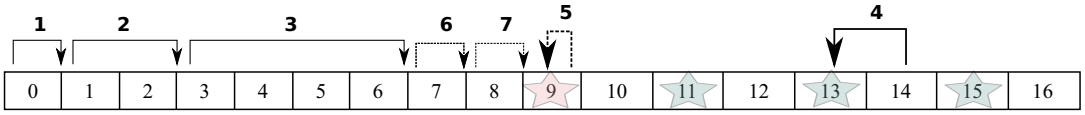


FIGURE 2.12 – Description de la procédure d'optimisation destructive-upper-bound.

2.4.4.3 Autres procédures

Les procédures bottom-up et dichotomic-bottom-up sont incomplètes si l'on impose une limite (temps, nœuds ...) sur la résolution de chaque sous-problème. On peut également limiter la « profondeur dichotomique » de dichotomic-bottom-up, c'est-à-dire le nombre maximal de divisions du domaine de la variable objectif.

2.5 Solveurs de contraintes

Nous citons quelques outils de programmation par contraintes libres ou commerciaux. Cette revue ne se veut pas exhaustive tant les outils disponibles évoluent continuellement. Dans un souci de concision, nous ne citerons que les solveurs dédiés aux problèmes booléens (SAT) et à la programmation mathématique mentionnés dans les chapitres suivants. Les outils présentés diffèrent en plusieurs points : les entités qu'ils sont capables de traiter (entiers, ensembles, objets ...) ; les contraintes et algorithmes de recherche implémentés ; le langage hôte sur lequel il s'appuie. Historiquement, les premiers solveurs étaient des extensions du langage déclaratif de programmation logique `Prolog` [21] dont l'algorithme d'unification est une clé. De nos jours, les extensions de Prolog les plus populaires sont `ECLiPSe` [22] et `SICStus Prolog` [23] qui proposent toutefois des interfaces vers des langages objet tels que Java, .NET, C et C++ ainsi qu'une compatibilité avec les langages d'autres solveurs. Les langages C et C++ ont connu beaucoup de succès grâce à leur rapidité et la gestion optimisée de la mémoire. Les noyaux des solveurs commerciaux `CHIP` [24] et `Ilog CP Optimizer` [25] sont principalement écrits en C/C++ mais leur architecture complexe comporte de nombreuses couches logicielles et interfaces dans d'autres langages. Les solveurs libres `gecode` [26] et `mistral` [27] entièrement écrits en C/C++ proposent des services classiques tout en restant modulaire et rapide. Il existe quelques solveurs libres implémentés en Java comme `choco` [28] et `koalog` [29] malgré des critiques concernant la gestion de la mémoire, notamment le ramasse-miettes (*garbage collector*), et les performances des programmes. Pour répondre à ces critiques, on peut d'abord remarquer que les performances des solveurs dépendent bien plus des technologies embarquées que du langage hôte. Ensuite, leur code source ouvert ainsi que la popularité et la portabilité du langage Java

en font des outils de choix pour la recherche et l'enseignement. Plus récemment, `comet` [30] a proposé un langage de modélisation orienté objet compréhensible par un solveur de contraintes proposant aussi des services de recherche locale et de programmation mathématique. Certains solveurs commerciaux tels `Ilog CP Optimizer` et `comet` proposent des fonctionnalités supplémentaires concernant le calcul distribué et parallèle. Dans un tout autre registre, le solveur `sugar` utilise une méthode d'*order encoding* [31] transformant un CSP vers un problème de satisfiabilité booléenne (SAT) qui est ensuite résolu efficacement par un des deux solveurs SAT simples `minisat` ou `picosat` Face à la diversité des solveurs et langages, un effort de standardisation des langages de modélisation a été récemment entrepris à l'instar de ce qui existe en programmation mathématique. Cependant, cet effort d'uniformisation des contraintes globales et des langages de description de CSP ne permettent pas encore de déclarer un problème sans aucune considération du solveur sous-jacent. Simultanément, il est encore nécessaire de développer des contraintes globales dédiées à la résolution d'un problème précis.

Au début de cette thèse, mon choix s'est naturellement porté sur le solveur `choco` développé au sein de l'École des Mines de Nantes pour bénéficier de l'expertise de mes collègues lors de mon apprentissage. La lecture et la modification du code source peuvent être un facteur clé de l'apprentissage et la compréhension tout en augmentant l'autonomie des utilisateurs chevronnés. Finalement, la participation à un projet dynamique et collaboratif a enrichi ma réflexion grâce aux rencontres, discussions et collaborations qui s'en sont suivies. Nous reviendrons en détail sur nos contributions au solveur Choco dans le chapitre 9 et en annexes A et B.

À titre d'exemple, le Listing 1 montre la déclaration et la résolution du modèle en figure 2.5 pour le problème des n reines avec l'API du solveur Choco. Une instance d'un CSP $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ est stockée dans un objet de la classe `CPModel` implémentant l'interface `Model`. On déclare d'abord les variables principales et auxiliaires stockées dans trois tableaux grâce aux fonctions de la fabrique `Choco.*` en précisant leurs domaines initiaux et leurs noms. On pose ensuite les contraintes de liaison et les contraintes globales dont les variables sont automatiquement extraites et ajoutées au modèle. On déclare ensuite un objet de la classe `CPSolver` implémentant l'interface `Solver` qui gère les structures de données nécessaires à la résolution. La lecture du modèle construit les objets utiles à la résolution en fonction d'options du modèle et de la structure du problème. Elle détermine entre autres les types des domaines (intervalle, énumération ...) et le filtrage des contraintes (consistance d'arcs, consistance de bornes ...). On lance ensuite l'exploration pour trouver toutes les solutions du problème avec la configuration par défaut (algorithmes de recherche, méthodes de séparation, heuristiques de sélection).

Listing 1 – Code source Java utilisant choco pour calculer toutes les solutions du problème des n reines décrit en figure 2.5.

```
Model m = new CPModel();
IntegerVariable[] queens = Choco.makeIntVarArray("Q", n, 1,n);
IntegerVariable[] p = Choco.makeIntVarArray("p", n, 1, 2*n);
IntegerVariable[] q = Choco.makeIntVarArray("q", n, -n, n);
for (int i = 0; i < n; i++) {
    m.addConstraints(eq(p[i], minus(queens[i], i)), eq(q[i], plus(queens[i], i)));
}
m.addConstraints(allDifferent(queens),allDifferent(p),allDifferent(q)));
Solver s = new CPSolver();
s.read(m); s.solveAll();
```

2.6 Conclusion

Nous avons décrit dans ce chapitre les principes de la Programmation Par Contraintes (PPC). Cette approche déclarative permet de résoudre des problèmes combinatoires variés. Les utilisateurs décrivent leur problème en posant des contraintes sur les valeurs que peuvent prendre les différentes variables composant le problème. Un solveur de contraintes est alors chargé de calculer les solutions du problème. Le processus de résolution exacte de problèmes de satisfaction de contraintes permet de générer efficacement les solutions d'un problème grâce à la combinaison des techniques de filtrage et des algorithmes de recherche.