

Des contraintes au programme

François Fages

L'informatique est probablement la technologie la plus connue et la science la plus insoupçonnée du public. Il ne se cache pourtant point de magie derrière l'affichage sur son ordinateur personnel des pages *web* du monde entier, ni derrière leur sélection intelligente par un moteur de recherche, ou la gestion des commandes de vol d'un avion, son atterrissage automatique, ou encore le calcul automatisé des emplois du temps des équipages en respect des règlements. Il ne se cache pas toujours non plus, un programme unique exécutant une suite d'opérations bien définie à l'avance dont on pourrait à coup sûr prédire le résultat. Il se trouve, à différentes échelles, des réseaux fixes ou dynamiques de programmes, développés depuis soixante ans par des ingénieurs et des chercheurs informaticiens, en même temps que leurs méthodes de conception, d'analyse et de vérification à l'aide... d'autres programmes.

La science informatique, née au début du siècle dernier de la logique mathématique et des besoins en machines à décoder des messages, est la *science du calculable*. Les ordinateurs sont des machines universelles, en principe capables d'effectuer tout calcul grâce à leur capacité à exécuter tout programme. Les programmes sont donc ce qui permet de transformer virtuellement un ordinateur en une infinité de machines différentes, dédiées à la réalisation d'autant de calculs particuliers, le temps de leur exécution. En ce sens, tous les ordinateurs sont équivalents, de même que tous les *langages de programmation*, dès lors qu'ils permettent d'exprimer tout calcul. Ils diffèrent cependant par les facilités qu'ils offrent pour décrire ces calculs de façon plus ou moins lisible et aisément vérifiable. Aujourd'hui encore, la création de programmes demeure un art, une tâche coûteuse qui demande créativité et ingéniosité, et de très grands efforts pour être garantie sans erreur. Le Saint Graal de la programmation est donc de s'en affranchir au profit d'une *modélisation* qui consiste à spécifier dans le programme uniquement les propriétés que la solution doit satisfaire, le « quoi », et non plus le « comment » qui est source d'erreurs. La programmation par contraintes est le style de programmation créé il y a trente ans qui se rapproche le plus de cette quête. Depuis vingt ans, elle s'applique avec succès dans l'industrie à la résolution de problèmes d'optimisation combinatoire de grande importance socio-économique ou écologique (recherche des meilleurs emplois du temps de personnels, des meilleurs ordres d'exécution des tâches à réaliser dans un chantier, des meilleures allocations de ressources, des meilleures découpes, empaquetages, etc.). Dans cet article nous tentons de dégager au travers d'exemples simples les aspects les plus fondamentaux de la programmation par contraintes et, nous l'espérons, les plus propices à une réflexion interdisciplinaire.

Le programme naît des contraintes

Dans un problème de Sudoku, on cherche à placer des chiffres de 1 à 9 sur une grille composée de neuf carrés de neuf cases, de telle sorte qu'aucune ligne, aucune colonne et aucun carré ne contienne deux fois le même chiffre. Initialement, certains chiffres sont placés et on cherche à compléter la grille en respectant ces contraintes. Par exemple, partant de la grille de la figure de gauche ci-dessous, on cherche à trouver la solution figurée à droite :

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

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

On peut modéliser un problème de Sudoku en associant à chaque case une variable mathématique, notée a_{ij} , où i désigne le numéro de ligne et j le numéro de colonne, prenant une valeur comprise entre 1 et 9, et en imposant que les contraintes suivantes soient satisfaites :

1. certaines valeurs a_{ij} sont fixées initialement (par exemple $a_{21}=7$);
2. a_{ij} est différent de a_{ik} dans chaque ligne i , pour toute paire d'indices j, k différents ;
3. a_{ij} est différent de a_{kj} dans chaque colonne j , pour toute paire d'indices i, k différents ;
4. a_{ij} est différent de a_{kl} dans chaque carré, pour tous les couples d'indices.

Dans un langage de programmation par contraintes, la modélisation précédente, saisie quasiment sous cette forme, constitue à elle seule le programme, et suffit à résoudre le problème en quelques millisecondes sur un ordinateur ordinaire. Elle définit complètement le problème, en ne spécifiant que le « quoi », et tout en laissant la méthode de résolution, le « comment », implicite.

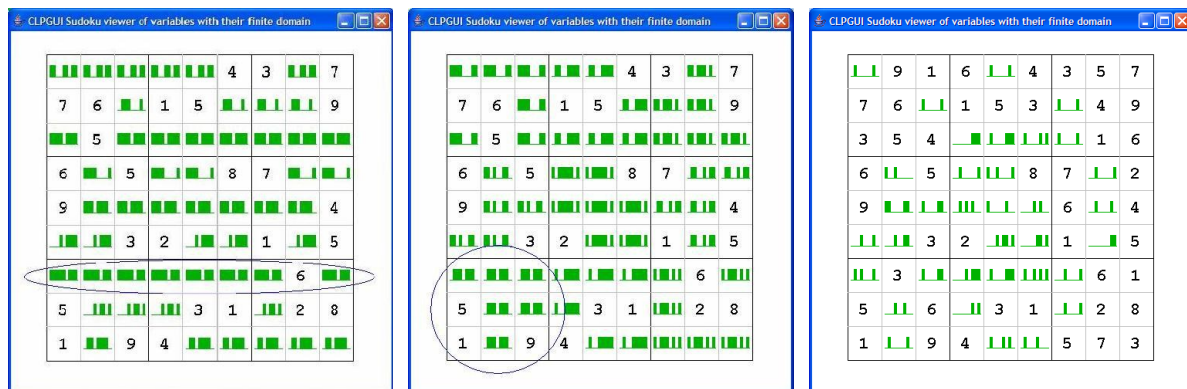
Il est à noter qu'une méthode naïve de résolution consisterait à énumérer toutes les combinaisons possibles des valeurs des variables a_{ij} et à tester à chaque fois si les contraintes sont satisfaites. Cependant un simple calcul montre qu'un tel algorithme ne terminerait pas son exécution en un temps raisonnable, car il devrait explorer dans le pire cas, un nombre exponentiel de configurations, 9^{81} , soit à raison d'un milliard de configurations par seconde, un temps d'exécution bien supérieur à l'âge de l'univers. Pour éviter cet écueil, l'idée de la programmation par contraintes est d'utiliser les contraintes de façon active pour réduire les domaines des variables avant et pendant l'énumération des valeurs possibles, et par là même réduire de façon drastique le nombre de possibilités à explorer.

Du point de vue de la modélisation du problème à résoudre, les contraintes sont des *formules logiques* qui expriment les relations devant être satisfaites entre les variables que l'on cherche à déterminer. Ces formules peuvent être arbitrairement compliquées. Elles s'interprètent dans des structures mathématiques, comme les nombres entiers dans l'exemple précédent, ou dans des structures de données quelconques, comme par exemple des assemblages de symboles permettant de représenter des formules logiques et donc ... des contraintes et des programmes. Pour cette raison, les programmes peuvent manipuler d'autres programmes, et notamment les traduire dans d'autres langages, et les exécuter. Par suite, toutes les machines programmables sont équivalentes. De même, tous les langages de programmation sont équivalents du strict point de vue des calculs qu'ils permettent de définir, et ne diffèrent que dans la façon plus ou moins commode qu'ils ont de les exprimer. Cette capacité d'autoréférence des programmes montre également que tout n'est pas calculable, car une telle puissance d'expression des langages de programmation s'accompagne alors de la possibilité de créer des paradoxes, comme celui du programme qui devrait déterminer si l'exécution d'un autre programme sur une donnée, tous deux fournis en entrée, se termine ou non. Un tel programme ne peut pas exister, car il serait alors très facile en lui rajoutant deux instructions, et en l'appliquant à lui-même, de créer un programme dont l'exécution se termine si et seulement si elle ne se termine pas. Ces

arguments sont bien sûr analogues au théorème d'incomplétude de Gödel qui montre qu'aucun système de preuve ne peut démontrer toutes les formules vraies des nombres entiers, à l'argument diagonal de Cantor qui montre que tous les infinis ne sont pas équivalents, et au paradoxe du menteur d'Epiménide le crétois qui montrait, il y a 2500 ans, qu'il est absurde qu'une personne dise qu'elle ment, tout comme le programme précédent supposé décider de la terminaison.

Le programme vit de luttes

Dans l'exemple du Sudoku, les contraintes binaires d'inégalité du type a_{ij} différent de a_{ki} sont utilisées dès que l'une des deux variables prend une valeur déterminée pour la retirer du domaine de l'autre variable. Par exemple, si l'on pose uniquement ces contraintes sur les lignes, sur les carrés, ou à la fois sur les lignes, les colonnes et les carrés, on obtient respectivement les domaines de valeurs possibles figurés ci-dessous dans chaque case par des traits (l'absence d'une valeur est matérialisée par un trou, par exemple la valeur 6 est retirée du domaine des variables de la ligne entourée dans la figure de gauche, tandis que les valeurs 1, 5 et 9 sont retirées du domaine des variables du carré entouré dans la figure du milieu) :

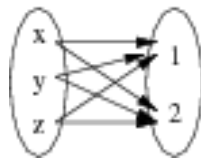


Cette lutte des contraintes pour rogner les domaines des variables, et donc les choix restants possibles, est proche du raisonnement humain. Dans beaucoup d'exemples de Sudoku, elle suffit à résoudre la grille. Cependant la dernière figure montre que l'action conjuguée des trois types de contraintes n'a pas suffi ici à placer automatiquement tous les chiffres. Dans cet exemple, le programme doit donc poursuivre la résolution en procédant par essais, c'est-à-dire en choisissant une variable indéterminée a_{ij} (par exemple a_{11}), en choisissant une valeur v dans son domaine (ici 2 ou 8), et en posant la contrainte $a_{ij}=v$. Cette valeur fixée est une nouvelle information utilisée par les autres contraintes pour à nouveau réduire les domaines des variables qui leurs sont reliées. En cas d'échec (lorsque le domaine d'une variable devient vide), la contrainte $a_{ij}=v$ est retirée ainsi que ses conséquences, et les autres valeurs sont essayées.

Du point de vue de l'exécution du programme, chaque contrainte peut donc être vue comme un *agent réactif* qui apporte de l'information à l'arrivée de nouvelles informations. Celles-ci proviennent des autres contraintes, que ce soit des contraintes initiales du problème à résoudre, ou bien des choix faits lors des essais de valeurs par la procédure de recherche énumérative avec laquelle l'action des contraintes se combine. Les calculs effectués par les contraintes peuvent cependant être arbitrairement compliqués. Toujours dans l'exemple du Sudoku, il y a par exemple un intérêt à traiter la contrainte *tous-différents* ($\{a_{i1}, \dots, a_{in}\}$) comme une seule *contrainte globale*, au lieu de la décomposer en contraintes binaires a_{ij} différent de a_{ik} , pour toute paire d'indices $j < k$. Par exemple, si l'on considère trois variables x, y, z , de domaine $\{1, 2\}$, la contrainte globale *tous-différents* ($\{x, y, z\}$) n'a pas de solution puisqu'il n'y a que deux

valeurs pour les trois variables, alors que ces deux valeurs suffisent à satisfaire chacune des contraintes binaires de sa décomposition, prises une à une. Cela se traduit par la détection immédiate de l'échec avec la contrainte globale, au lieu d'une poursuite, coûteuse en temps de calcul, de l'exploration de l'arbre de recherche avec les contraintes décomposées.

Le raisonnement sur les contraintes globales peut alors faire appel à des méthodes de calcul spécifiques qui dépassent le raisonnement humain. Par exemple, la contrainte globale *tous-différents* ($\{x_1, \dots, x_n\}$) peut utiliser un calcul efficace de couplage dans un graphe biparti pour tester l'existence de solution et réduire les domaines des variables. Dans cette représentation, les ensembles des variables et des valeurs forment respectivement les deux catégories de sommets du graphe biparti, et les flèches associent aux variables les valeurs qu'elles peuvent prendre. Dans le précédent exemple de trois variables pouvant prendre deux valeurs, le graphe biparti associé est le suivant :



Dans cette représentation, notant n le nombre de variables et k le nombre de valeurs, la contrainte globale *tous-différents* peut être satisfaite si et seulement s'il existe un couplage, c'est-à-dire un sous-ensemble de n flèches n'ayant aucune extrémité en commun avec d'autres flèches. Or pour effectuer ce calcul la théorie des graphes nous fournit un algorithme très efficace, demandant un faible nombre d'opérations (de l'ordre de k fois la racine carrée de n) en fonction de la taille du problème (k et n), au lieu de l'énumération des valeurs possibles qui nécessiterait un nombre exponentiel d'opérations (de l'ordre de k^n) impossible à exécuter en un temps raisonnable. Cet algorithme détecte immédiatement l'absence de solution dans l'exemple précédent. Comme c'est souvent le cas pour les contraintes globales, la conception d'un algorithme de résolution efficace est passée par un changement de représentation du problème.

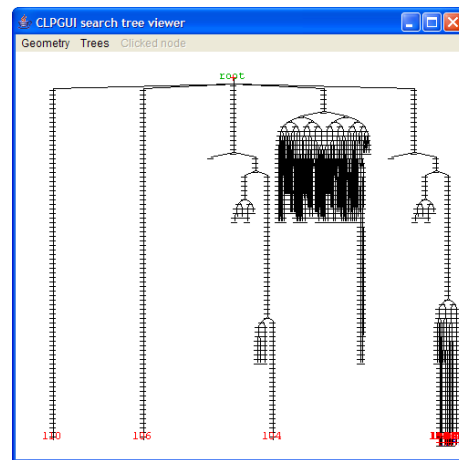
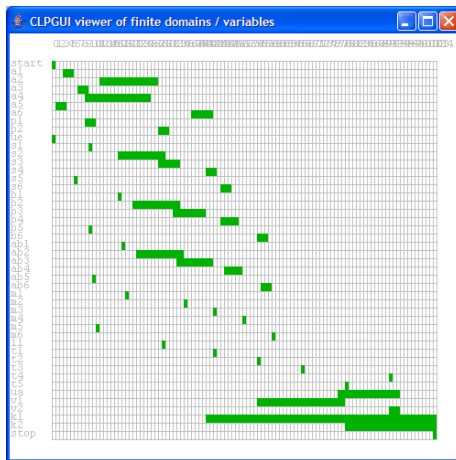
La recherche des solutions à un problème de satisfaction de contraintes est ainsi une lutte des contraintes contre l'explosion combinatoire due à l'énumération. Chaque contrainte participe à la réduction des domaines des variables et à la détection des échecs, en repoussant l'exploration énumérative des valeurs possibles au moment où plus aucune information ne peut être déduite. Cette exploration se décrit par un *arbre de recherche* dans lequel les sommets représentent les points de choix (par exemple les variables indéterminées) et les flèches les choix à essayer (par exemple les valeurs possibles). L'exploration de cet arbre de possibilités ne peut en général être évitée mais les contraintes contribuent à réduire la taille de l'arbre en éliminant les branchements impossibles en chaque sommet. Les problèmes de Sudoku se résolvent de cette manière généralement sans recherche ou avec un très petit nombre de points de choix. Dans d'autres problèmes, la taille des arbres de recherche développés peut être substantielle, que ce soit pour trouver une solution, ou pour prouver qu'il n'existe pas de solution. Ce dernier cas se présente notamment dans les problèmes d'*optimisation*, pour lesquels on ne cherche pas juste une solution, mais la « meilleure » solution vis-à-vis d'un critère de coût, ce qui revient à montrer de plus qu'il n'existe pas de solution de coût inférieur à celui de la meilleure solution trouvée.

Par exemple dans un *problème d'ordonnancement* pour la construction d'un pont, l'on s'intéresse à déterminer les ordres d'exécution des tâches qui permettent d'achever la construction le plus rapidement possible. Les tâches de creusement puis coulage des fondations et élévation des piliers doivent se faire dans un ordre bien précis pour un même pilier, mais sans contrainte d'ordre entre les différents piliers. Par ailleurs, les tâches utilisant une même

ressource comme une grue par exemple, ne peuvent pas se réaliser en même temps. Dans un tel problème d'ordonnancement, chaque tâche a une durée dx connue, et une date de démarrage x que l'on cherche à déterminer en fonction des contraintes et du critère d'optimisation. Le programme se résume logiquement aux contraintes suivantes :

1. $x+dx$ inférieur à y si la tâche démarrant à la date x doit précéder la tâche démarrant à la date y ;
2. $x+dx$ inférieur à y ou bien $y+dy$ inférieur à x si les deux tâches sont mutuellement exclusives (car elles utilisent une même ressource par exemple) ;
3. minimiser la date de fin de la dernière tâche.

Les contraintes de précédence (dans 1.) luttent pour la réduction des domaines par un simple calcul de borne sur les variables : la contrainte $x+dx$ inférieur à y est vue comme un agent qui impose en permanence que la borne maximum de x soit plus petite ou égale à la borne maximum de y moins dx lorsque cette borne diminue, et réciproquement que la borne minimum de y soit plus grande ou égale à la borne minimum de x plus dx lorsque cette borne augmente. Les contraintes disjonctives (dans 2.) donnent lieu à l'exploration d'un arbre de recherche binaire dont chaque sommet correspond à un choix de précédence. La figure de gauche ci-dessous montre une solution prouvée optimale à un problème d'ordonnancement disjonctif de 46 tâches. Celles-ci sont figurées sur les différentes lignes, avec leurs dates possibles de démarrage indiquées dans les colonnes. Certaines tâches critiques ont une date de démarrage fixe trouvée par le programme de façon à assurer l'optimalité, et sont figurées par un point, tandis que d'autres tâches non critiques ont une certaine flexibilité et peuvent démarrer dans un intervalle de temps calculé par le programme, représenté dans la figure de gauche ci-dessous par un segment horizontal :



Plusieurs solutions optimales, ayant des flexibilités différentes, peuvent exister et être énumérées. La figure de droite montre l'arbre de recherche exploré pour effectuer l'optimisation (contrainte 3.). Les premières branches à gauche dans l'arbre correspondent aux recherches successives de solutions, avec à chaque fois l'ajout d'une contrainte imposant de trouver une solution de coût strictement meilleur. Dans cette *lutte pour l'optimalité* cette fois, trois solutions sont ainsi trouvées de coût respectivement, 110, 106 et 104. Ensuite l'arbre de recherche relativement dense qui suit, ne comporte que des échecs car aucune branche n'arrive à son terme, et constitue à lui seul la *preuve d'optimalité* de la dernière valeur de coût 104. Enfin les solutions optimales sont énumérées en fixant le coût à la valeur prouvée optimale 104. Dans cet arbre, chaque noeud correspond au choix d'une alternative dans une contrainte d'exclusion mutuelle. Pour cet exemple, incalculable à la main, la résolution totale n'aura ainsi exploré que 5000 nœuds et pris 50 millisecondes, preuve d'optimalité comprise.

Le programme meurt de liberté

Le nombre d'atomes dans l'univers, 10^{80} , ne représente que le nombre de libres choix d'un chiffre parmi dix par quatre-vingt personnes. L'exploration d'un tel espace de recherche n'a donc de sens que si elle peut être circonscrite par les contraintes d'un problème à résoudre. La difficulté d'un problème dépend alors de la capacité ou non des contraintes devant être satisfaites à diriger la recherche de la réponse. Pour cette raison, la difficulté d'un problème est relativement indépendante de la taille *a priori* de son espace de recherche. Par exemple, ordonner une suite de n éléments par ordre croissant revient à calculer une suite parmi l'ensemble de toutes les permutations possibles, soit parmi, n fois $n-1$ fois ... fois 2, noté $n!$, permutations. Or ce problème est résolu trivialement par des algorithmes de tri nécessitant un nombre très limité d'opérations, de l'ordre de n fois $\log(n)$, où $\log(n)$ est le nombre de décimales de n (par exemple 80 pour le nombre d'atomes dans l'univers). Cette complexité de calcul pour trier n éléments est d'ailleurs optimale, car $n \log(n)$ est du même ordre de grandeur que $\log(n!)$ qui est égal au nombre minimum de tests nécessaires pour discriminer une suite parmi $n!$ suites possibles : aucun algorithme de tri ne pourrait donc effectuer moins d'opérations. En revanche, déterminer si une conjonction de n disjonctions de k variables booléennes (prenant deux valeurs possibles vrai ou faux) ou leur négation peut être satisfaite, a un espace de recherche constitué des 2^k valeurs possibles des variables, et on ne connaît pas pour ce problème d'algorithme faisant moins d'opérations dans le pire cas, c'est-à-dire arrivant dans tous les cas à éliminer le balayage des valeurs une à une.

Ce dernier problème, nommé SAT, est le prototype des problèmes *non déterministes polynomiaux*, dits NP. Par définition, un problème est dans la classe de complexité NP s'il existe une procédure permettant, pour toute donnée du problème fournie en entrée, de trouver une solution quand il en existe, en faisant des opérations et des *choix* quelconques mais en nombre restreint, borné par un polynôme en fonction de la taille de la donnée. Par exemple SAT est dans NP parce que pour trouver une solution s'il en existe, il suffit de choisir k valeurs de vérité pour les variables, et de vérifier pour chaque disjonction si elle est vraie, ce qui nécessite également k opérations au plus. En 1970, Stephen Cook a montré de plus que tout problème dans NP peut être transformé en un problème SAT équivalent de taille polynomiale. Par exemple, le problème du Sudoku généralisé à des grilles de taille n quelconque et n valeurs, peut être codé dans SAT en associant à chaque case n variables booléennes, une pour chaque valeur, et en transcrivant les contraintes *tous-différents* en des contraintes disjonctives sur les variables booléennes. Un algorithme pour SAT permet donc de résoudre en principe tout autre problème dans NP. Le problème SAT est ainsi représentatif des problèmes les plus difficiles dans NP, appelés NP-complets.

Pour ces raisons, la classe de complexité NP-complet caractérise les problèmes difficiles que l'on résout en procédant par essais. Elle s'oppose à la classe P des problèmes considérés comme faciles, que l'on peut résoudre avec un algorithme *déterministe* effectuant un nombre restreint d'opérations, borné lui aussi par un polynôme en fonction de la taille de l'entrée, mais ne faisant *aucun choix*. Le problème du Sudoku généralisé à des grilles de taille quelconque a été montré NP-complet. Il en va de même pour les problèmes d'ordonnement disjonctif de la section précédente. En revanche, les problèmes d'ordonnement sans contrainte d'exclusion mutuelle (contrainte 2.) sont dans P, comme d'ailleurs le montre le programme de la section précédente puisqu'il ne crée pas de point de choix dans ce cas.

Un problème NP-complet peut néanmoins être facile à résoudre sur certaines données, et certaines applications pratiques de problèmes NP-complets peuvent très bien être résolues efficacement en programmation par contraintes pour des données de relativement grande taille,

alors que sur d'autres données de relativement petite taille, le programme se perdra dans l'exploration d'un nombre astronomique de points de choix. Dans le cas de SAT, ces différences de comportement ont été analysées en fonction de la *densité en contraintes* des problèmes, définie comme la valeur du nombre de contraintes divisé par le nombre de variables. Les problèmes de faible densité sont généralement faciles à résoudre car il est aisé d'exhiber une solution. Les problèmes de haute densité sont également faciles à résoudre car il est alors aisé d'exhiber une contradiction dans les contraintes permettant de conclure qu'il n'existe pas de solution. Cependant il existe une valeur moyenne de densité autour de laquelle se trouvent les problèmes les plus difficiles à résoudre. Dans SAT, un phénomène étonnant de *transition de phase*, signifiant un passage abrupte entre satisfiabilité et insatisfiabilité, a été montré mathématiquement autour d'une densité voisine de 4,3 : lorsque la taille n des problèmes tend vers l'infini, les problèmes SAT de densité inférieure à cette valeur tendent à être tous satisfiables, tandis que les problèmes de densité supérieure tendent à être tous insatisfiables. Trop de contraintes tuent les contraintes, et pas assez tue le problème. C'est également autour de cette densité de 4,3 que sont observés expérimentalement les plus grands temps de calcul des programmes résolvant le problème SAT.

Enfin, certaines explosions combinatoires dans le nombre de choix à explorer peuvent être dues à la présence de *symétries* dans le problème. Par exemple, si plusieurs variables x_1, \dots, x_n jouent des rôles symétriques, on divise par $n!$ le nombre de possibilités en ajoutant une contrainte d'ordre entre ces variables pour casser la recherche des solutions symétriques. De même, si certaines permutations de valeurs ont des rôles symétriques, on peut casser ces symétries en ajoutant des contraintes d'ordre entre les variables et les variables d'indice permuté. Les combinaisons de symétries de variables et de valeurs peuvent être cassées de la même manière par l'ajout combiné des deux types de contraintes d'ordre. Cependant le nombre de symétries peut lui-même être astronomique, et le mur de complexité se retrouver alors dans leur élimination.

Conclusion

« L'art naît de contraintes, vit de luttes et meurt de liberté » écrivait Gide, à la suite de Michel-Ange. Dans l'art de la programmation par contraintes, le programme naît des contraintes, vit de leur lutte pour réduire l'espace de recherche, et meurt des points de choix laissés libres. La beauté de la programmation par contraintes réside dans sa concision à décrire le « quoi » et non le « comment », au travers d'une modélisation du problème à résoudre par des variables mathématiques et des contraintes. L'efficacité de son exécution sur ordinateur dépend de l'utilisation active des contraintes pour réduire les domaines des variables et éliminer les choix dans l'espace de recherche. La maîtrise ou non de la complexité du problème modélisé sous forme de contraintes logiques, dépend de leur capacité à déduire les zones de passage obligé dans l'univers des choix possibles.

Mais au final, pourrait-on éliminer totalement l'exploration non déterministe des alternatives, en nombre exponentiel, par un calcul déterministe de complexité polynomiale fournissant toujours une solution ou la garantie qu'il n'en existe pas? Pourrait-on résoudre tout Sudoku par un calcul sans recherche? Ces questions renvoient à la plus importante conjecture de l'informatique énoncée par Stephen Cook dès 1970 : $NP \neq P$, les problèmes que l'on peut résoudre en temps non déterministe polynomial ne peuvent pas tous se résoudre en temps déterministe polynomial. L'expérience nous indique bien que les problèmes dans NP sont de toute évidence plus difficiles à résoudre que ceux dans P, que les calculs non déterministes sont plus puissants que les calculs déterministes, que le libre choix l'emporte sur l'âne de Buridan, cependant à ce jour personne n'a réussi à le démontrer mathématiquement.

