Optimisation combinatoire : comment trouver rapidement une bonne solution ?

Philippe Morignot, CEA LIST, LRI, philippe.morignot@cea.fr

Qu'y a-t-il de commun entre construire un emploi du temps, pour que des tâches soient effectuées par des employés d'un aéroport par exemple, et s'amuser à résoudre une grille de sudoku dans son journal favori ?

Dans le premier cas, la direction de l'aéroport dispose d'avions d'un côté et de ses employés de l'autre : chaque avion doit être servi selon des tâches précises par un employé, à certaines heures précises et avec une certaine durée, après l'atterrissage de l'avion et avant qu'il ne décolle à nouveau. Et, d'un autre côté, la journée de travail de chaque employé est régie par le Code du Travail, les accords syndicaux et les accords propres à l'entreprise (un aéroport, dans notre exemple). Par exemple, un employé ne peut pas effectuer deux tâches en même temps. Ou bien, il faut laisser quelques minutes libres entre deux tâches successives pour que chaque employé puisse se rendre du lieu de la première tâche au lieu de la deuxième, dans sa journée de travail. Comment faire correspondre travail à effectuer sur chaque avion et respect des contraintes juridiques pour chaque employé ?

Dans le deuxième cas, on doit placer un nombre de 1 à 9 dans chaque case vide de la grille de sudoku, de façon qu'aucun nombre n'apparaisse 2 fois dans une même ligne, une même colonne, ou sur chaque petite grille intérieure de 9 cases. L'aide fournie par le concepteur de la grille se résume à avoir placé certains nombres dans certaines cases. Comment donc trouver les nombres qu'il faut placer dans les cases vides ?

Le premier problème est pratique et doit être résolu tous les jours par la direction de tous les aéroports du monde, pour qu'ils puissent accueillir des passagers et leur faire prendre des avions. Le deuxième problème est un jeu disponible dans de nombreux journaux, pour amuser le lecteur. Que peut-il donc y avoir de commun entre deux problèmes aussi disparates ?

Le point commun réside dans la résultat d'un petit calcul, que nous allons détailler pour chacun de ces deux cas, et dont nous allons tirer les conséquences.

Evaluons à gros grain le nombre d'emploi du temps que l'on pourrait construire pour nos employés d'aéroport. Imaginons qu'il y ait 600 tâches à effectuer par jour, pour 150 employés disponibles. Une première tâche pourrait être affectée à disons 100 employés : ceux dont la journée de travail commence avant le début de cette tâche et finit après la fin de cette tâche (on suppose que 50 employés ne répondent pas à ce critère pour cette tâche là). Une deuxième tâche, qui chevauche la première, pourrait être affectée à 99 autres employés : toujours 100 personnes qui peuvent effectuer cette tâche sans faire d'heures supplémentaires, moins l'employé précédent, qui est déjà pris. Une troisième tâche pourrait être affectée aux 98 employés restants. Et ainsi de suite, jusqu'à la centième tâche. Soit 100 x 99 x 98 x 97 x ... x 3 x 2 x 1 = 100 ! (lire : factorielle 100) possibilités.

Maintenant, il paraît raisonnable de supposer qu'un employé effectue plusieurs tâches pendant sa journée de travail, ce qui veut dire que certaines tâches à placer ne se chevauchent pas. On recommence ainsi le même calcul précédent pour la deuxième centaine de tâche. Puis pour les quatre autres centaines. On arrive ainsi à 6 x 100 ! possibilités, ce qui fait $5{,}60 * 10^{158}$ possibilités, pour une évaluation très grossière, encore une fois.

Effectuons le même calcul pour le jeu de sudoku : Pour une case vide d'une petite grille intérieure de sudoku, il y a a priori 8 nombres de 1 à 9 qui peuvent convenir : les 9 nombres possibles moins 1 nombre, celui qui se trouve déjà dans la petite grille (en suppose qu'il n'y ait qu'un seul nombre déjà placé par petite grille intérieure). Pour une deuxième case vide, il ne reste plus que non pas 8 nombres mais 7 : les 8 précédents, moins celui qui vient d'être placé. Puis, pour encore une autre case vide, il ne reste plus que 6 nombres (8 moins les 2 précédents). Et ainsi de suite, jusqu'à ce qu'il ne reste plus qu'une seule possibilité pour le nombre de la dernière case de chaque petite grille.

Pour résumer, il y a donc 8 x 7 x 6 x 5 x 4 x 3 x 2 x 1 = 8 ! (lire : factorielle 8) possibilités pour remplir chaque petite grille de sudoku. Et comme il y a typiquement 9 petites grilles juxtaposées par grille, il y a donc $(8\ !)^9$ possibilités de remplir un sudoku entier. Cela fait $2{,}82 * 10^{41}$ possibilités.

Considérons un instant ces deux nombres, 6 x 100 ! possibilités pour un emploi du temps et $(8\ !)^9$ possibilités pour un sudoku. Ces nombres sont déjà très grands, il est inenvisageable d'essayer à la main toutes ces possibilités, pour trouver un emploi du temps ou une grille de sudoku.

On pourrait penser faire explorer toutes ces possibilités par un programme sur un ordinateur, ce qui accélèrerait déjà nettement les choses par rapport à l'approche manuelle, avec un papier et un crayon ! Mais cela ne fait que repousser le problème : que se passerait il si on augmentait la taille du problème ? C'est à dire, au lieu de chercher à résoudre un sudoku de taille 9 x 9, chercher à résoudre un sudoku 100 x 100, ou 1000 x 1000 ? Et, au lieu de chercher à affecter 600 tâches à 150 employés, chercher à en affecter 6000 à 1500 employés, ou même 60 000 à 15 000 employés ?

La réponse est que le nombre de possibilités à explorer devient démesurément trop grand, même pour le plus rapide des ordinateurs au monde : il suffit d'augmenter un peu la taille des données du problème pour que le nombre de possibilités augmente démesurément trop pour être parcourues une à une.

C'est le phénomène d'explosion combinatoire : il y a beaucoup trop de possibilités envisageables pour un problème pour que même un ordinateur puisse les considérer en un temps raisonnable. Ce que l'on aimerait bien, c'est avoir une solution en quelques secondes, voire quelques minutes, sur un ordinateur, et certainement pas de le laisser tourner des milliards d'années pour avoir une solution à notre problème !

C'est là le domaine de l'optimisation combinatoire : le nombre de combinaisons à envisager a priori pour trouver une solution à ces problèmes est démesurément trop grand pour pouvoir être parcouru une à une, même pour l'ordinateur le plus rapide au monde. Si de petits problèmes peuvent quand même être résolus de cette manière, augmenter la taille de l'une des données du problème se traduit par une augmentation correspondante démesurée du nombre de possibilités, qui en pratique interdit d'utiliser la même méthode.

Il faut donc faire autrement qu'essayer toutes les possibilités. Comment ?

Une première idée consiste à effectivement parcourir toutes les solutions une à une, mais dans un ordre intéressant. C'est à dire qui mène à une solution en ayant parcouru relativement peu de possibilités, en espérant bien ne pas avoir à les parcourir toutes (ce qui n'est pas

raisonnablement possible). Ce mode de parcours s'effectue au moyen d'une heuristique : mener à une solution mieux que ne le ferait le hasard, au moyen d'une information non disponible dans les données du problème (information exogène). Trouver une telle information est très difficile : si l'on en disposait pour tout problème (ce qui s'appelle un oracle), il suffirait de s'en servir pour tomber rapidement sur la solution lors de notre parcours exhaustif. Aussi, un problème ne disposant pas d'oracle, il faut bâtir une heuristique, plus elle se rapprochera de l'oracle (il faut là aussi le montrer, puisque cet oracle n'est justement pas connu), plus rapide sera la convergence vers une solution. Avec, toujours, le risque de ne pas converger du tout (et dans ce cas nous revoilà reparti pour des milliards d'années de calcul pour effectuer un parcours de toutes les possibilités…).

Maintenant, un parcours intelligent doit exploiter la structure du problème posé. Cette structure contient des informations précieuses qu'il serait judicieux d'exploiter pour augmenter l'intelligence de ce parcours.

Cette structure peut se représenter en langage informatique comme des relations qui doivent toujours être vérifiées entre les variables du problème. Par exemple, pour le jeu de sudoku, le fait que deux nombres de deux cases vides soient différents (les 2 variables correspondant à chacune de ces cases vides sont différentes). Ou que les nombres d'une ligne soient exactement les nombre de 1 à 9 (toutes ces variables sont différentes deux à deux). Ces relations sont des contraintes, qui traduisent cette structure et ce mode résolution est la programmation par contraintes.

Ce mode de résolution de problèmes combinatoires a été proposé par Jean-Louis Laurière il y a 35 ans. Depuis, on trouve de nombreux packages se basant sur ces idées, notamment au dessus de Java ou basés sur Prolog.

Une autre façon d'aborder cette exploration consiste passer de solutions incomplètes ou partielles à d'autres solutions incomplètes ou partielles, en partant d'une solution très incomplète (le problème initial juste posé) jusqu'à espérer tomber sur une solution du problème posé. Il s'agit alors de parcourir un graphe d'états où un état est une solution partielle du problème posé. Par exemple, pour en revenir au jeu de sudoku, on peut poser un état comme une configuration donnée de nombres dans les cases vides. Et on progresse d'un état à un autre état en changeant les nombres des cases vides, par permutation de 2 nombres par exemple.

Ce mode de résolution de problèmes combinatoires a donné lieu à de nombreux algorithmes (les algorithmes de recherche dans un espace d'états), dont l'algorithme de Dijkstra à la fin des années 50, et l'algorithme A* proposé par Peter Hart, Nils Nilsson et Berttram Raphael à la fin des années 60.

Pour en savoir plus :

Irène Charon, Anne Germa, Olivier Hudry, Méthodes d'optimisation combinatoire, Masson, Collection pédagogique de télécommunication, 1996.

Stuart Russell, Peter Norvig. Artificial Intelligence : A Modern Approach. Prentice Hall, New Jersey, 2010, 3rd edition. Chapitres 3 et 6.