Sudokus et programmation linéaire

par Renaud Sirdey*

À Benoît.

Nous illustrons l'omniprésence de la programmation linéaire en optimisation combinatoire à travers une application ludique d'actualité : la résolution de Sudokus.

Introduction

Dans un précédent article [6], nous avons appliqué la méthode du recuit simulé au problème du complètement d'une grille de Sudoku. Dans l'optique, à nouveau, d'illustrer l'art de concevoir des algorithmes de résolution de problèmes combinatoires à travers cette application ludique, nous présentons une méthode basée sur la programmation linéaire.

I La programmation linéaire

L'invention, en 1947, de la programmation linéaire, indissociable de celle de l'algorithme du simplexe, par G.-B. Dantzig est l'un des tours de force mathématiques du vingtième siècle.

Formellement, un programme linéaire consiste, étant donnés un vecteur $c \in \mathbb{R}^n$, une matrice A, $m \times n$, à valeurs réelles, et un vecteur $b \in \mathbb{R}^m$, à trouver un vecteur $x \in \mathbb{R}^n$, solution du programme mathématique suivant :

$$\begin{cases} \text{Minimiser } c^T x, \\ \text{sous la contrainte} \\ Ax \leq b. \end{cases}$$

Les premières applications de la programmation linéaire ont concerné les plans de formation, d'approvisionnement et de déploiement des troupes de l'armée de l'air américaine. Ces plans, appelés *programmes*, ont donné leur nom à la programmation linéaire puis, peu après, à la programmation mathématique en général [2].

L'exemple probablement le plus classique d'application de la programmation linéaire consiste à déterminer le régime alimentaire le moins cher qui soit conforme aux recommandations des nutritionnistes [1]. Soit D un ensemble de denrées alimentaires et N un ensemble de nutriments (glucides, protides, lipides, etc.). On note c_d le coût unitaire de la denrée d et A_{nd} la quantité unitaire du nutriment n apportée par la denrée d. Enfin, b_n^- et b_n^+ dénotent respectivement les apports quotidiens minimum et maximum recommandés. Soit x_d la quantité de la denrée d à acheter, le problème s'énonce alors comme suit

$$\begin{cases} \text{Minimiser } \displaystyle\sum_{d \in D} c_d x_d, \\ \text{sous les contraintes} \\ b_n^- \leq \displaystyle\sum_{d \in D} A_{nd} x_d \leq b_n^+ & \forall n \in N, \\ x_d \geq 0 & \forall d \in D. \end{cases}$$

Contrairement aux apparences, un programme linéaire est un problème d'optimisation combinatoire. En effet, un tel programme possède une interprétation géométrique assez intuitive : dans la mesure où chaque ligne de la matrice A définit, avec le coefficient du vecteur b associé, un hyperplan, le problème revient à optimiser une forme linéaire sur un polyèdre, l'optimum étant forcément réalisé en l'un de ses sommets. Puisque ces derniers sont en nombre fini, il s'agit bien d'un problème de nature combinatoire.

Généralement, on résout les programmes linéaires à l'aide de l'algorithme du simplexe. Grossièrement, cet algorithme part d'un sommet du polyèdre puis, à chaque itération, emprunte l'une de ses arêtes de manière à atteindre un meilleur sommet voisin. Ce schéma est répété jusqu'à ce que le sommet courant soit au moins aussi bon que tous ses voisins, donc optimal. Bien que son efficacité pratique soit remarquable, l'algorithme du simplexe n'est pas un

* renauds@nortel.com

algorithme polynomial : il existe des instances pathologiques qui requièrent un nombre exponentiel d'itérations [1].

Néanmoins, il existe des algorithmes capables de résoudre le problème de la programmation linéaire en temps polynomial, en particulier l'algorithme de l'ellipsoïde introduit en 1979 par L.G. Khachiyan. Bien qu'il ne soit paradoxalement pas capable de rivaliser, en pratique, avec l'algorithme du simplexe, ce dernier algorithme est d'une importance théorique capitale : il fournit un outil précieux utilisé pour établir, de manière quelque peu indirecte, le caractère polynomial de certains problèmes combinatoires (nous n'entrerons pas plus dans les détails ici, voir par exemple [4]).

II Contraintes d'intégrité

Un grand nombre de problèmes d'optimisation combinatoire peuvent être mis sous la forme d'un programme linéaire en nombres entiers, c'est-à-dire d'un programme mathématique de la forme

$$\begin{cases} \text{Minimiser } c^T x, \\ \text{sous les contraintes} \\ Ax \leq b, \\ x \in \mathbb{Z}^n. \end{cases} \quad (1)$$

Bien que les contraintes d'intégrité ne changent pas la nature du problème (il s'agit toujours d'optimiser une forme linéaire sur un polyèdre[1]), le formalisme ci-dessus permet d'exprimer de nombreux problèmes d'optimisation *NP*-difficiles.

Il suit qu'à moins que $P = NP$, il n'existe pas d'algorithme permettant de résoudre le problème de la programmation linéaire en nombres entiers en temps polynomial.

Il y a deux principales stratégies d'approche d'un problème *NP*-difficile, qu'il soit donné sous la forme d'un programme linéaire en nombres entiers ou non.

Soit on cherche à le résoudre *de manière exacte* à l'aide d'un algorithme dont la complexité est *exponentielle*, en espérant que les temps de calcul s'avéreront acceptables sur la majorité des instances.

Soit on cherche à le résoudre *de manière approchée* à l'aide d'un algorithme dont la complexité est *polynomiale*, en espérant que les solutions ainsi obtenues seront de qualité acceptable sur la majorité des instances.

Que ce soit en résolution exacte ou approchée des programmes linéaires en nombres entiers associés à des problèmes *NP*-difficiles, la relaxation linéaire, c'est-à-dire le programme linéaire continu obtenu en ignorant les contraintes d'intégrité (1), joue souvent un rôle de premier plan. En particulier, une approche actuellement parmi les plus performantes pour la résolution exacte de tels problèmes consiste à exploiter, dans le cadre d'un schéma de recherche arborescente, une relaxation linéaire dynamiquement enrichie de contraintes spécialisées qui définissent des facettes (faces propres de dimension maximale) de l'enveloppe convexe des solutions entières (voir [7] pour un exemple).

Dans la suite, nous nous proposons de chercher à résoudre le problème du complètement d'une grille de Sudoku, qui est *NP*-complet[2] ([8]), de manière approchée mais en temps polynomial.

III Résolution de Sudokus

Commençons par mettre le problème de résolution d'une grille de Sudoku sous la forme d'un programme linéaire en 0-1.

Pour ce faire, nous introduisons les 9^3 variables

$$x_{ijk} = \begin{cases} 1 & \text{si la case } ij \text{ contient le chiffre } k \\ 0 & \text{sinon} \end{cases},$$

pour $i \in \{1,2,\ldots,9\}$, $j \in \{1,2,\ldots,9\}$ et $k \in \{1,2,\ldots,9\}$.

Tout d'abord, chaque case ne doit contenir qu'une et une seule valeur, d'où les contraintes

$$\sum_{k=1}^{9} x_{ijk} = 1,$$

pour tout $i \in \{1,2,\ldots,9\}$ et tout $j \in \{1,2,\ldots,9\}$.

Il convient ensuite de s'assurer que les règles du jeu sont respectées. Chaque ligne doit contenir chaque valeur une et une seule fois, d'où les contraintes

$$\sum_{j=1}^{9} x_{ijk} = 1,$$

pour tout $i \in \{1,2,\ldots,9\}$ et tout $k \in \{1,2,\ldots,9\}$. Il en est de même pour chacune des colonnes, d'où les contraintes

$$\sum_{i=1}^{9} x_{ijk} = 1,$$

pour tout $j \in \{1,2,\ldots,9\}$ et tout $k \in \{1,2,\ldots,9\}$. De plus, on a les contraintes

$$\sum_{i=1}^{3}\sum_{j=1}^{3} x_{3i_0+i,3j_0+j,k} = 1,$$

[1] Généralement défini, cette fois, par un système linéaire d'inégalités de taille astronomique.

[2] Un problème *NP*-difficile est le pendant « optimisation » d'un problème de *décision NP*-complet. Ici, il s'agit de décider si un Sudoku (de taille arbitraire) peut être complété ou non ; nous avons montré dans [6] comment associer un problème d'optimisation à ce problème de décision.

pour tout $i_0 \in \{0,1,2\}$ et tout $j_0 \in \{0,1,2\}$, qui permettent de s'assurer que chaque valeur est présente une et une seule fois dans chacun des sous-carrés.

Enfin, il convient de fixer les variables associées aux cases fixes. On note F l'ensemble des triplets (ligne, colonne, valeur) qui définissent ces cases.

L'encadré suivant donne l'énoncé du programme linéaire en nombres entiers résultant[3]. Ce programme possède $9^3 = 729$ variables et $4 \times 81 = 324$ contraintes (contraintes de types (2) et (3) non comprises).

Énoncé du programme linéaire en nombres entiers

$$
\begin{cases}
\text{Trouver } x \\
\text{sous les contraintes} \\
\displaystyle\sum_{k=1}^{9} x_{ijk} = 1 & i,j \in \{1,2,\ldots,9\}, \\
\displaystyle\sum_{j=1}^{9} x_{ijk} = 1 & i,k \in \{1,2,\ldots,9\}, \\
\displaystyle\sum_{i=1}^{9} x_{ijk} = 1 & j,k \in \{1,2,\ldots,9\}, \\
\displaystyle\sum_{i=1}^{3}\sum_{j=1}^{3} x_{3i_0+i,3j_0+j,k} = 1 & i_0, j_0 \in \{0,1,2\}, \\
 & k \in \{1,2,\ldots,9\}, \\
x_{ijk} = 1 & \{i,j,k\} \in F, \quad (2) \\
x_{ijk} \in \{0,1\} & i,j,k \in \{1,2,\ldots,9\}. \quad (3)
\end{cases}
$$

La plupart des logiciels généralistes de résolution de programmes linéaires en nombres entiers optent pour l'approche exacte (par recherche arborescente plus ou moins sophistiquée), donc en temps exponentiel. Remarquons par ailleurs que la formulation de l'encadré se prête bien à l'utilisation de tels logiciels ([5]). Ceci fournit une première illustration de la pertinence pratique de cette formulation car il est relativement rare que l'on puisse directement donner un programme linéaire en nombres entiers difficile à ce type de logiciel.

Ici, nous nous proposons d'exploiter la *relaxation linéaire* du programme ci-dessus, c'est-à-dire le programme linéaire où l'on a remplacé les contraintes (3) par les contraintes $0 \leq x_{ijk} \leq 1$ ($i,j,k \in \{1,2,\ldots,9\}$), afin d'essayer de compléter des grilles de Sudoku par résolution d'une séquence de programmes linéaires continus.

Notre algorithme s'exprime comme suit.

Tout d'abord, on résout la relaxation linéaire du programme de l'encadré. Si, par chance, la solution est entière, on a alors fini.

Dans le cas contraire, on répète le schéma itératif suivant.

On commence par trier les variables par distance décroissante à l'intégrité (ici $|x_{ijk} - 0.5|$), les variables sont ensuite considérées dans cet ordre. Soit x_{ijk} la variable courante, si fixer x_{ijk} à 1 rend le programme linéaire irréalisable, alors on fixe x_{ijk} à 0 (*i.e.* on ajoute la contrainte $x_{ijk} = 0$), on résout à nouveau la relaxation linéaire et on recommence, sinon si fixer x_{ijk} à 0 rend le programme linéaire irréalisable, alors on fixe x_{ijk} à 1 (*i.e.* on ajoute la contrainte $x_{ijk} = 1$), on résout à nouveau la relaxation linéaire et on recommence, sinon on passe à la variable suivante.

On arrête ce schéma itératif soit lorsqu'une solution entière est obtenue, autrement dit lorsque l'on a résolu le problème, soit lorsque l'on n'a pas pu fixer de variable supplémentaire.

En raison de ce dernier critère d'arrêt, l'algorithme peut ne pas trouver la solution. Il s'agit donc bien d'un algorithme de résolution approchée. Qui plus est, dans la mesure où le problème est *NP*-complet et où l'algorithme ci-dessus est polynomial (en effet, seule est requise la résolution d'un nombre polynomial de programmes linéaires continus, ce que l'on peut théoriquement faire en temps polynomial à l'aide de l'algorithme de l'ellipsoïde) on sait, sous l'hypothèse que $P \neq NP$, qu'il ne peut résoudre le problème dans tous les cas.

IV Mise en œuvre et résultats empiriques

Nous avons mis en œuvre l'algorithme de la section précédente à l'aide de COIN-OR[4], un environnement logiciel libre pour l'optimisation, conçu et développé principalement par des chercheurs du centre Thomas J. Watson d'IBM. Entre autres, cet environnement fournit une implémentation particulièrement performante de l'algorithme du simplexe (librairie CLP).

En pratique, notre algorithme fonctionne remarquablement bien. Par exemple, pour les quatre Sudokus « diaboliques » utilisés dans notre précédent article [6], il ne faut respectivement fixer que 2, 3, 3 et 1 variables pour obtenir la solution. L'algorithme du

[3] Bien que l'auteur l'ait retrouvée indépendamment, il convient d'attribuer la paternité de cette formulation à T. Koch [5], qui s'y attaque directement à l'aide d'un logiciel de résolution de programmes linéaires en nombres entiers.

[4] *COmputational INfrastructure for Operations Research* (`www.coin-or.org`).

simplexe est respectivement exécuté 21, 7, 7 et 3 fois. Le temps d'exécution est égal à 2,3, 1,0, 1,0 et 0,6 secondes[5] respectivement. Par ailleurs, la relaxation linéaire est entière pour le Sudoku « diabolique » donné dans [3], une seule exécution de l'algorithme du simplexe est donc requise et le temps d'exécution est de 0,3 secondes.

Nous avons essayé l'algorithme sur d'autres Sudokus de différents niveaux de difficulté et, jusqu'à présent, la solution a toujours été obtenue en quelques itérations (typiquement après fixation de moins de cinq variables). En particulier, l'algorithme n'a pour l'instant jamais été mis en échec !

Conclusion

Il est assez rare que la programmation linéaire puisse être exploitée de manière aussi directe pour résoudre un problème combinatoire *NP*-complet. Néanmoins, les résultats de la section précédente ne doivent pas masquer le fait que notre méthode reste une « bonne » heuristique : bien que cela semble peu fréquent, il est plus que vraisemblable que l'algorithme ne soit pas en mesure de résoudre certaines instances.

Références

[1] V. Chvátal, *Linear programming*. A Series of Books in the Mathematical Sciences, W.H. Freeman and Company, 1986.

[2] G.B. Dantzig, « Reminiscences about the origins of linear programming », *Operations Research Letters* **1** (1982) 43–48.

[3] J.-P. Delahaye, « Le tsunami du Sudoku », *Pour la Science* (Déc. 2005).

[4] M. Grötschel, L. Lovász et A. Schrijver, *Geometric algorithms and combinatorial optimization*, tome 1 de *Algorithms and Combinatorics*, Springer, 1988.

[5] T. Koch, « Rapid mathematical programming or how to solve Sudoku puzzles in a few seconds », Rapport technique ZIB-05-51, Konrad-Zuse-Zentrum für Informationstechnik Berlin, Décembre 2005.

[6] R. Sirdey, « Sudokus et algorithmes de recuit », *Quadrature* **62** (Oct.–Déc. 2006) 9–13.

[7] R. Sirdey et H. Kerivin, « A branch-and-cut algorithm for a resource-constrained scheduling problem ». Rapport technique PE/BSC/INF/017633 V01 EN, Service d'architecture BSC, Nortel GSM Access R&D, France, *RAIRO Oper. Res.* (2006) (à paraître).

[8] T. Yato, *Complexity and completeness of finding another solution and its application to puzzles*, MSc thesis, Université de Tokyo, 2003.

[5] Mesures réalisées sur un ordinateur portable des plus moyens.