Couper, attendrir, trancher, réduire :

un conte culinaire sur la résolution informatique des problèmes difficiles

Nicolas Schabanel, *directeur de recherches CNRS à l'Université Paris Diderot*
Pierre Pansu, *professeur à l'Université Paris-Sud*

Comment répartir ses invités sur deux tablées sans placer ensemble ceux qui se détestent ? Résoudre ce casse-tête quel que soit le nombre d'invités est un problème extrêmement difficile à resoudre quand le nombre d'invités augmente. À l'aide d'un peu de programmation géométrique, on peut trouver une réponse approchée de garantie présumée optimale.

Ce soir, je convie mes amis à dîner. Cependant, je ne dispose pas d'une table assez grande et je vais devoir les répartir sur deux tables. Quels groupes former ?

Fermement décidé à résoudre ce problème de façon rationnelle, j'attrape un crayon et mon bloc-note et appelle Maelys, toujours la première informée des derniers potins, pour savoir qui s'est fâché avec qui dernièrement. Tout en tenant le téléphone d'une main, je me retrouve à représenter chaque invité par un petit rond (on dit un *sommet* en théorie des graphes) et à tracer un trait (une *arête*, c'est plus chic) entre chaque paire d'invités qu'il vaudrait mieux séparer en ce moment. Je me rends rapidement compte que la situation est assez compliquée : il y a beaucoup plus d'animosités entre eux que je ne le croyais ; par exemple, aucun groupe ne semble se détacher clairement sur mon dessin (on dit un *graphe*).

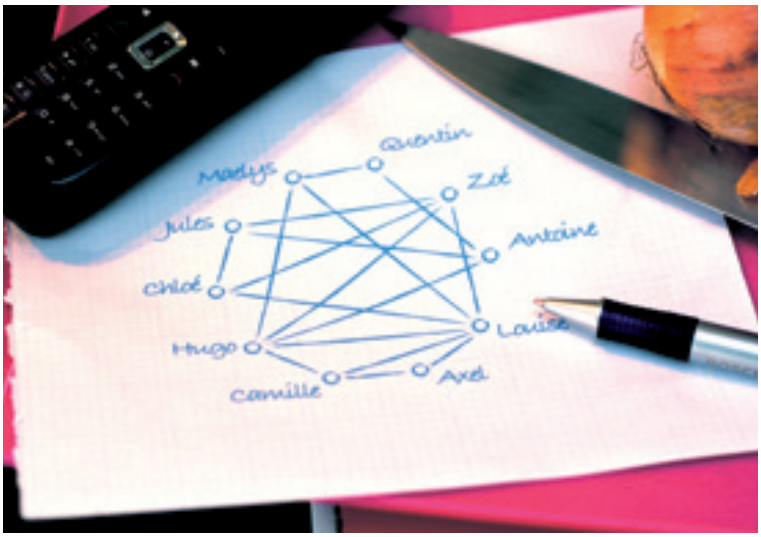

Figure 1. Le graphe des inimitiés entre mes amis

Je vais donc essayer de trouver la répartition de mes amis en deux groupes qui minimise le nombre d'inimitiés à l'intérieur de chaque table, ce qui revient à maximiser le nombre d'arêtes allant d'une table à l'autre. Mathématiquement, je cherche à *partitionner* les sommets de mon graphe en deux parties de sorte à maximiser le nombre d'arêtes cassées par cette partition, c'est-à-dire à cheval entre les deux tables. (Pour ne pas compliquer inutilement, nous ne chercherons pas à obtenir deux parties de même taille. Les techniques de résolutions avec ou sans cette contrainte ne diffèrent d'ailleurs pas significativement.)

Quand est-ce qu'on mange ?

Si j'étais physicien, j'aurais certainement monté le dispositif suivant : pour chaque invité je prends une bille, et pour chaque arête je place un gros ressort comprimé simulant l'inimitié entre les deux billes ; puis je lâche le tout dans un couloir, les ressorts écartent les billes fâchées qui s'étalent sur toute la longueur du couloir, et j'obtiens mon plan de table en coupant le couloir au milieu : je répartis les invités sur les deux tables en fonction du bout du couloir où a atterri leur bille. Le problème est que je ne disposerais alors d'aucune garantie sur la qualité de la solution que j'aurais ainsi trouvée.

Un mathématicien prouverait très facilement que l'on peut trouver une solution optimale. En effet, j'ai deux choix possibles pour chaque invité : le placer à la première ou à la seconde table. Il n'y a donc que $2 \times 2 \times \ldots \times 2 = 2^n$ façons possibles de répartir les n invités sur les deux tables. Il me suffit donc de toutes les énumérer et de garder celle qui coupe le maximum d'arêtes. Cependant, cette méthode vaut-elle mieux que la précédente ?

Certes, elle trouvera la solution optimale, mais au bout d'un temps *exponentiellement* long : 2^n essais. Par exemple, avec 60 invités, il faudrait énumérer et tester 2^{60} soit environ 10^{18} partitions, ce qui prendrait approximativement 10^9 secondes soit à peu près 100 ans sur un hypothétique processeur surpuissant cadencé à 100 GHz. Et si l'on invitait 61 personnes, ce serait 200 ans qu'il faudrait attendre, 400 ans pour 62, etc. Autant dire que cette solution n'en est pas une dès lors que le nombre d'invités augmente un peu : ajouter un seul invité double le temps de calcul (et ce *indépendamment* de la vitesse du processeur) ! Inversement, doubler la vitesse du processeur ne permet de résoudre une situation qu'avec un invité de plus seulement !

Avec 60 invités, il faudrait énumérer et tester 2^{60} soit environ 10^{18} partitions, ce qui prendrait approximativement 10^9 secondes soit à peu près 100 ans sur un hypothétique processeur surpuissant cadencé à 100 GHz.

Réduire et tomber sur un os

Peut-on trouver la meilleure solution à notre problème dans un temps raisonnable ? En tant qu'informaticien, je vous répondrai que nous avons de bonnes raisons de penser que

Méthodes efficaces et inefficaces

En informatique, le consensus traditionnel est de considérer qu'une méthode de résolution est *efficace* si le nombre d'opérations à effectuer est *au plus polynomial* en fonction de la taille du problème (au plus n^{α} pour un problème de taille n, avec α quelconque). Pour une méthode polynomiale, doubler la taille du problème augmente le temps de résolution d'un facteur constant $2^{\frac{1}{\alpha}}$: les variations du temps de calcul « suivent » celles de la taille des données dans des proportions bornées.

À l'opposé, l'effet de la taille sur un algorithme de résolution en temps exponentiel (par exemple 2^n) est catastrophique :

le temps de calcul d'un algorithme en temps exponentiel double chaque fois que la taille du problème augmente de un seulement ! ($2^{n+1} = 2 \times 2^n$). Par exemple, si la puissance de calcul des ordinateurs double tous les 18 mois, alors, avec un algorithme polynomial, on peut espérer résoudre des problèmes $2^{\frac{1}{\alpha}}$ fois plus grands tous les 18 mois, alors qu'avec un algorithme exponentiel, on ne peut espérer résoudre que des problèmes de taille +1 tous les 18 mois…

Le nombre d'opérations en fonction de la taille n de l'entrée du problème à résoudre est une mesure *intrinsèque* de la complexité d'un problème : *indépendamment de la machine utilisée*, plus cette fonction croît rapidement, plus il sera difficile de résoudre ce problème en grande taille.

ce n'est pas gagné car ce problème (dont le nom scientifique est *Max-Cut*) est « un des plus durs ». Voici ce que je veux dire par là.

Pour résoudre un problème avec un ordinateur, il faut lui donner une recette (le *programme*) qui le résolve et ce quelles que soient les valeurs des paramètres (ici : le nombre d'invités et la liste des arêtes d'inimitié entre eux). Pour écrire cette recette (je veux dire, ce *programme* ; on parle aussi d'*algorithme*), on peut l'écrire de toutes pièces, ou bien s'appuyer sur d'autres recettes afin d'éviter de perdre trop de temps (ou risquer d'introduire de nouveaux bugs). Par exemple, on peut imaginer que la recette qui résoudrait le problème hypothétique *A* = « sortir ce robot de cette pièce » pourrait s'appuyer sur la recette *B* = « trouver la porte de cette pièce » et qu'une fois la porte trouvée il est relativement simple d'y aller pour sortir. Puisqu'on sait comment résoudre *A* facilement si on sait résoudre *B*, cela démontre que le problème *A* est *moins difficile* que *B*. C'est ainsi que les informaticiens démontrent qu'un problème *A* n'est pas plus dur qu'un autre *B* : en supposant connue une recette pour *B* et en écrivant alors une recette efficace pour *A* qui utilise celle de *B*. On dit alors que la résolution de *A se réduit* à celle de *B*.

Or, il se trouve que notre problème de placement d'invité (*Max-Cut*) fait parti du club des problèmes auxquels on peut réduire n'importe quel problème « *raisonnable* » : si vous me fournissez une recette efficace à notre problème Max-Cut, je peux vous écrire une recette qui résout n'importe quel problème « raisonnable ». Cela veut dire que parmi les problèmes que l'on pourra résoudre en un temps « raisonnable », Max-Cut est l'un des plus durs ! On dit que Max-Cut est *NP-complet* (voir l'encadré *Les problèmes les plus durs : les problèmes NP-complets*).

La grande question ouverte en informatique est : est-ce que l'on pourra un jour trouver

P =? NP : *la mécanisation de l'intuition est-elle possible ?*

Une question fondamentale soulevée par l'informatique est exprimée par la formule

P =? NP

qui demande s'il existe une méthode de résolution efficace pour tous les problèmes pour lesquels on peut *vérifier* efficacement la solution une fois qu'on la connaît.

P est l'ensemble des problèmes pour lesquels il existe une construction efficace de la solution (voir l'encadré *Méthodes efficaces et inefficaces*). Pour de nombreux problèmes importants (tout particulièrement pour l'industrie), aucun algorithme efficace n'est connu actuellement. Par exemple, sur les ordinateurs actuels, aucune méthode connue ne trouve efficacement le chemin le plus court passant par toutes les villes d'un pays et retournant à son point de départ avec plus de 1500 villes.

Pourtant, pour la plupart de ces problèmes importants, il est très facile de vérifier efficacement qu'une solution apportée par quelqu'un est effectivement correcte ou non (ces problèmes sont dits NP). Par exemple, on ne connaît aucune méthode efficace pour trouver un coloriage d'une carte avec trois couleurs de sorte que deux pays voisins aient des couleurs différentes, mais cette propriété est très facile à vérifier sur une proposition de coloriage. Cela signifie que pour les problèmes NP, si l'on a d'abord la bonne *intuition* sur ce que devrait être la solution, on peut ensuite vérifier efficacement que la solution est effectivement correcte.

La question P =? NP demande donc si tous les problèmes NP admettent une solution efficace. Elle résume ainsi la question philosophique « l'intuition est-elle mécanisable » ? Cette question figure parmi la liste des sept problèmes mathématiques pour le prochain millénaire sélectionnés par l'institut Clay en 2000 et dont la résolution serait récompensée d'un million de dollars. La réponse, aujourd'hui inconnue, aurait un retentissement bien au-delà du seul cadre de l'informatique.

Les problèmes les plus durs : les problèmes NP-complets

La plupart des problèmes de calcul à résoudre dans la vie réelle appartiennent à la classe NP, ceux où l'intuition peut fonctionner (voir l'encadré *P =? NP : la mécanisation de l'intuition est-elle possible ?*).

Certains problèmes NP sont-ils plus difficiles ou plus importants que d'autres, au sens où ils concentreraient toute la difficulté des problèmes NP ? En 1971, de part et d'autre du mur de Berlin, Stephen A. Cook et Leonid A. Levine ont démontré qu'il existait un problème dans NP qui était effectivement plus difficile que tous les autres, au sens que tous les autres problèmes de NP s'y ramènent : s'il existe une méthode de résolution efficace pour celui-ci, on peut la transformer en une méthode de résolution efficace pour tous les problèmes de NP !

Un tel problème est dit NP-*complet*. Richard M. Karp a démontré l'année suivante qu'en fait de très nombreux problèmes de NP qui n'avaient a priori rien à voir (coloration d'une carte, voyageur de commerce…) sont tous NP-complets ! La résolution d'un seul d'entre eux entraîne la résolution de tous les autres ! C'est par exemple le cas du problème Max-Cut étudié ici.

La plupart des nombreux problèmes étudiés en informatique sont soit dans P, soit NP-complets, et rares sont ceux qui échappent à cette dichotomie – parmi eux cependant un problème emblématique : la factorisation des grands entiers, clé de voûte de la sécurité sur Internet actuellement avec le protocole RSA (voir l'article p. 83).

Malgré des efforts importants, aucune avancée déterminante n'a été encore accomplie indiquant l'existence ou l'impossibilité d'une résolution efficace d'aucun de ces problèmes NP-complets. Vu la diversité de ces problèmes, la plupart des chercheurs pensent qu'il n'existe pas de moyen efficace pour résoudre les problèmes NP-complets.

une solution efficace pour le club des problèmes les plus durs ? La plupart des chercheurs pensent que non, mais nous n'avons pas de réponse claire pour l'instant (voir l'encadré *P =? NP : la mécanisation de l'intuition est-elle possible ?*).

Attendrissons notre problème…

En attendant que cette question trouve sa réponse, il faudrait tout de même que l'on arrive à placer nos invités. Si le problème est trop dur, peut-être qu'en assouplissant quelques contraintes, nous pourrions le

rendre plus sympathique ! Par exemple, nous pourrions nous satisfaire d'une recette qui garantisse de couper au moins 99 % du nombre d'arêtes coupées par la solution optimale. Malheureusement, on peut démontrer qu'approcher à 16/17 soit 94,1 % le nombre d'arêtes coupées optimal fait lui aussi partie du « club des problèmes les plus durs » et n'a donc probablement pas de recette efficace.

Si le problème est trop dur, peut-être qu'en assouplissant quelques contraintes, nous pourrions le rendre plus sympathique !

Essayons donc d'assouplir encore les contraintes. Pour cela, nous allons utiliser une forme originale de programmation : la programmation géométrique. Un des problèmes principaux avec l'approche physicienne évoquée au départ est que les billes et les ressorts ont beaucoup de mal à circuler dans le couloir trop étroit et se gênent les unes les autres, ce qui complique énormément la convergence vers un état d'équilibre optimal. Si l'on se donnait un peu plus d'espace, peut-être que l'on pourrait assurer une convergence efficace du dispositif vers un état d'équilibre optimal et même certifier la qualité de la solution produite.

Un gros oignon et quelques clous de girofle

Voici comment nous allons procéder. Au lieu de les représenter par des billes, nous allons représenter les n invités par autant de vecteurs de taille 1 (on dit *unitaires*) attachés à une origine commune, ceci dans un espace à n dimensions (cela veut juste dire que nous choisissons de donner n composantes à chaque vecteur, afin qu'il ait de la place pour bouger – notre bille d'avant n'avait qu'une seule dimension pour se déplacer : sa position dans le couloir, c'était trop contraignant). Comme pour les billes, nous souhaitons éloigner deux vecteurs correspondant à des invités reliés par une arête d'inimitié. Au lieu de relier ces vecteurs par des ressorts comprimés, nous allons chercher à les faire pointer dans des directions opposées. Pour cela, nous allons utiliser le fait que le *produit scalaire* entre deux vecteurs est positif s'ils pointent dans la même direction et négatif dans le cas contraire, et même d'autant plus négatif qu'ils pointent dans des directions opposées. Nous recherchons alors à disposer les vecteurs de sorte de *minimiser la somme des produits scalaires* entre les vecteurs correspondant à des invités liés par une arête d'inimitié. Or,

Figure 2. Une projection sur un oignon tridimensionnel de nos n vecteurs n-dimensionnels dans la configuration optimale déterminée par nos contraintes sur les produits scalaires.

il se trouve que l'on dispose d'une recette (un programme) pour résoudre *efficacement* ce genre de contraintes portant sur des produits scalaires entre n vecteurs n-dimensionnels, c'est la « programmation semi-définie » généralisant la « programmation linéaire ».

Tranchons dans le tas

Grâce à notre recette, nous avons maintenant obtenu une configuration de nos n vecteurs unitaires n-dimensionnels qui minimise la somme des produits scalaires des vecteurs correspondant aux invités fâchés. Pour les répartir en deux groupes, il me suffit alors de trancher par un plan (en fait on dit un *hyperplan* en dimension n) passant par l'origine des vecteurs, dans une direction choisie *au hasard*: les invités seront sur l'une ou l'autre des tables selon que leurs vecteurs tombent de l'un ou l'autre côté du plan.

Le choix d'une direction aléatoire pour trancher est un ingrédient essentiel de la recette. On peut aussi procéder sans aléatoire, mais c'est un peu plus compliqué et repose paradoxalement de manière cruciale sur l'analyse de la méthode aléatoire. Michel Goemans et David Williamson ont démontré en 1994 que cette solution coupe toujours au moins 87,8 % du nombre optimal d'arêtes. J'ai donc rempli ma mission avec une marge d'erreur relativement faible, garantie inférieure à 12,2 %.

Coupe toujours

De nombreuses variantes de ce problème existent : on peut chercher à couper en deux parties égales, en k morceaux, ou bien ajouter des poids sur les arêtes pour pondérer les inimitiés entre individus : la solution proposée ci-dessus peut être adaptée.

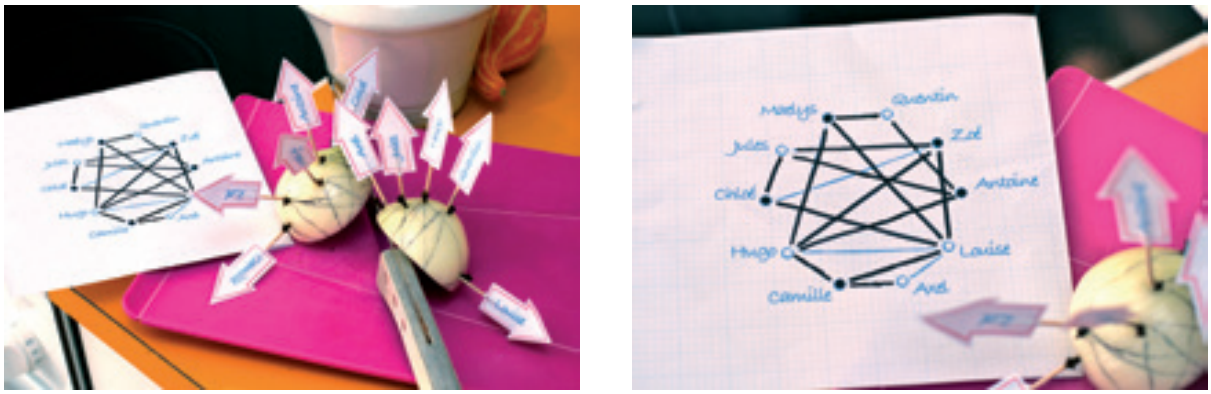

Figure 3. Une fois tranchée suivant une direction aléatoire, nous obtenons la partition des invités ci-dessus, qui coupe effectivement un grand nombre d'inimitiés (en noir), certifié en moyenne supérieur à 87,8 % de l'optimum !

Les problèmes de coupes dans les graphes ont de très nombreuses applications dans la vie pratique. On peut citer par exemple la détection de *communautés* dans les graphes dont les applications sont multiples : en sociologie, où on pourra isoler des groupes d'amis (ayant peu d'inimitié entre eux) ; en marketing, où on pourra suggérer à une personne un achat via son appartenance à tel profil de consommation ; dans la conception de micro-processeurs, où on pourra regrouper les composants en minimisant les câbles qui se croisent…

Pour conclure, on peut voir ici un très joli lien entre une géométrie très *continue* (je peux déplacer mes vecteurs petit à petit dans l'espace) et un problème purement discret (les solutions possibles sont clairement séparées les unes des autres, il n'y a pas d'intermédiaire possible entre placer un invité sur une table ou sur l'autre). Bien que l'informatique soit le royaume du discret (tout y est codé sous la forme d'entiers représentés en mémoire par des suites finies de bits, 0 ou 1), il semble que les interactions entre continu et discret ne se limitent pas à l'obtention de bonnes approximations. En effet, des résultats récents établissent des connexions intrigantes entre des conjectures sur la difficulté de résoudre informatiquement certains problèmes et certaines conjectures purement géométriques.

Ces problèmes ont de très nombreuses applications dans la vie pratique : en marketing, où on pourra suggérer à une personne un achat via son appartenance à tel profil de consommation ; dans la conception de micro-processeurs, où on pourra regrouper les composants en minimisant les câbles qui se croisent…

Mais je me rends compte que mes invités arrivent et qu'il est temps de passer à table avant de risquer une indigestion mathématico-informatique !