Structures de données des mathématiques discrètes

Arbre généalogique ou plus généralement organisation des informations sous forme arborescente ? Emploi du temps du lycée ? Meilleur itinéraire GPS ? Optimisation du trafic Internet sur des réseaux mobiles ? Modélisation de relations entre des personnes compatibles entre elles ou non ? Validation d'une procédure chirurgicale décomposée en étapes et actions permettant de passer d'une étape à l'autre ? Simulation d'un réseau de neurones biologiques ou analyse d'un génôme ? Tous ces mécanismes mettent en jeu des algorithmes qui reposent sur un objet mathématique et informatique essentiel : le graphe.

Mathématiquement, c'est un objet riche de propriétés algébriques. Informatiquement, il y a différentes manières de représenter un graphe, selon l'algorithme considéré. Et de très grands algorithmes de l'informatique, puissants et bien utiles, travaillent justement sur ces graphes. Découvrons tout cela.

Un graphe : un modèle des nœuds et des arcs

Un graphe est un ensemble de points, dont certaines paires sont reliées par des liens.

Un graphe est un ensemble d'unités, dont certaines paires sont reliées par des connections. Des neurones par des synapses, des ordinateurs par des connections réseaux, etc..

Un graphe est un ensemble d'îlots, certains reliés entre eux deux à deux par, par exemple, sept ponts.

Un graphe est un ensemble de ponts, certains liés entre eux par le fait qu'ils jouxtent le même ilôt.

Un graphe est un ensemble d'atomes, reliés entre eux par des liaisons chimiques, pour former un modèle de molécule.

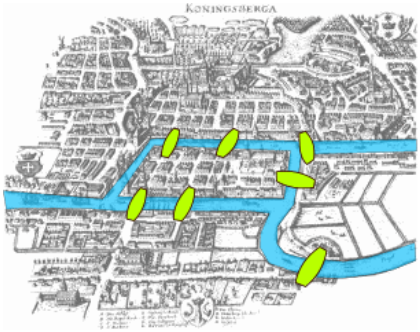
Un graphe est un ensemble d'objets informatiques, dont certains sont connectés entre eux par des références.

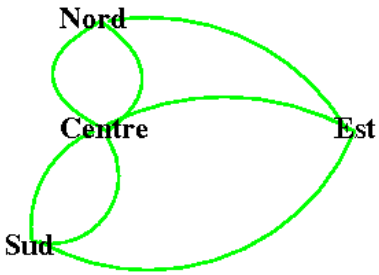
Un graphe est un ensemble dénombrable d'éléments, dont certaines paires sont reliées par une relation binaire.

Un graphe est un ensemble de *nœuds*, donc certains sont connectés entre eux par des *bords* : des *arêtes* (si le sens de la connection importe peu) ou des *arcs* (si la connection est orientée).

Et il s'avère que . . toutes ces définitions correspondent à la même abstraction : un graphe.

Si la notion de graphe est déjà utilisée dès l'antiquité pour décrire graphiquement ou formellement des relations entre des entités, ou expliquer des façons de jouer à des jeux de société, on accorde aisément au mathématicien suisse Leonhard Euler, d'avoir "inventé" le graphe en présentant en 1735, le problème des sept ponts de Königsberg, schématisé ci-dessous :





Le problème consistait à trouver une promenade à partir d'un point donné qui fasse revenir à ce point en passant une fois et une seule par chacun des ponts. Un chemin passant par chaque arête exactement une fois est un *chemin eulérien*, ou *circuit* du même nom s'il finit là où il a commencé.

L'*ordre* d'un graphe est le nombre de nœuds. Le *degré* d'un nœud est son nombre d'arêtes. Euler avait formulé qu'un graphe n'a un cycle eulérien que si chaque nœud est de degré pair, donc a un nombre pair d'arêtes. Cela fut démontré 130 ans plus tard. Ici ce qui distingue profondément la vision mathématiques de la vision informatique des graphes, sera, de ne passer de résultats existentiels (prouver si il existe (ou pas) tel objet) à des résultats effectifs : la fourniture du mécanisme certifié pour calculer l'objet.

Chaque fois qu'un problème se pose en terme d'une énumération d'éléments avec des relations entre ces éléments, nous sommes quasiment certains que nous allons pouvoir le modéliser sous forme de graphe. Ce graphe peut-être aussi *probabiliste*, c'est-à-dire que le passage d'un nœud à l'autre se fait par un arc caractérisé par une probabilité de transition.

Un graphe : un objet avec des méthodes permettant d'accéder aux éléments

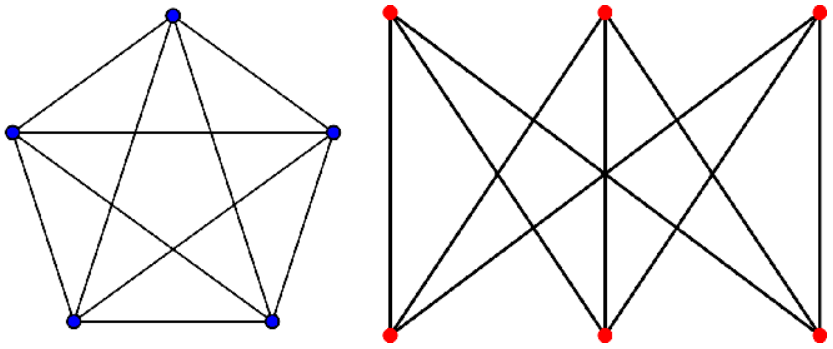
Informatiquement, un graphe G est un objet que l'on peut construire avec les méthodes :
`add_node(graph` G`, node_value` x`) ;`
qui permet d'ajouter un nœud x ou
`add_edge(graph` G`, node_value` x`, node_value` y`, edge_value` v`) ;`
qui permet d'ajouter un bord de x avec y avec une valeur v, tandis que les méthodes :
`del_node(graph` G`, node_value` x`) ;`
et
`del_edge(graph` G`, node_value` x`, node_value` y`, edge_value` v`) ;`
permettent de détruire les éléments correspondants, et les méthodes :
`node_value[] get_nodes(graph` G`) ;`
permettent de retrouver tous les nœuds du graphe, et :
`node_value[] get_edges(graph` G`, node_value` x`, node_value` y`) ;`
tous les bords entre deux nœuds.

Pour mémoriser ces éléments on peut utiliser une *liste d'adjacence*, c'est-à-dire la liste de nœuds destination de chaque nœud source :
`node_value[] get_nodes(graph` G`, node_value` x`) ;` En lien avec l'algèbre linéaire nous pouvons stocker la *matrice d'adjacence* : un tableau à deux dimensions, dans laquelle les lignes et les colonnes représentent respectivement les nœuds source et destination, les entrées dans la matrice indiquant quel bord existe entre les nœuds associés à cette ligne et colonne. La *matrice d'incidence* est celle dans laquelle les lignes représentent les nœuds et les colonnes représentent les bords, les entrées indiquant si les deux sont liés, à savoir incidents. Ces structures de données permettent de mettre en oeuvre des algorithmes efficaces pour calculer sur ces graphes les propriétés souhaitées.

Un *arbre* est un graphe non orienté, acyclique et connexe. Tout graphe non orienté sans cycles dont le nombre de nœuds excède le nombre de bords d'une unité exactement est un arbre. Les feuilles de l'arbre sont les nœuds de degré 1, relié à un seul autre nœud, tous les autres nœuds sont internes. Il y a, une fois les nœuds fixés, n^{n-2} arbres à n nœuds de topographie différente.

Si l'on choisit arbitrairement un nœud interne comme racine r il est possible d'enraciner l'arbre en r, c'est-à-dire orienter toutes les arêtes de sorte qu'il existe un chemin de r à tous les autres nœuds. Il permet alors de représenter par exemple des hiérarchies. Un nœud a alors des parents et des enfants.

Un arbre est un exemple de *graphe planaire*, c'est-à-dire qui a la particularité de pouvoir se représenter sur un plan sans qu'aucun bord n'en croise un autre. On peut facilement caractériser un graphe planaire. Un graphe fini est planaire si et seulement s'il ne compte pas parmi ces "sous-graphes" (c'est une notion assez fine que nous ne détaillerons pas ici) les deux graphes suivants :



Voici les quelques notions qui vont nous permettre de découvrir ce qui peut se calculer avec un graphe.

Le calcul d'un plus court chemin

Parmi les algorithmes les plus intéressants sur les graphes, c'est certainement celui du plus court chemin qui vient en premier. Attribuant à chaque bord une valeur, il s'agit de trouver le chemin le plus court d'un nœud donné à un autre nœud du graphe. Pour un tel problème nous pourrions redouter une explosion

combinatoire : à chaque nœud il y a un "carrefour" et il faut essayer tous les bords possibles et ainsi de suite, avec une augmentation multiplicative des possibilités.

Cependant choisir une structure de donnée pertinente permet non seulement de réduire cette complexité, mais aussi de résoudre le problème dans toute sa généralité. C'est le très bel algorithme de Roy-Warshall-Floyd qui illustre le mieux ce principe. Il s'agit ici de calculer, pour tout nœud x et tout nœud y, la valeur minimale des chemins de x à y, notée $L(x, y)$. C'est un algorithme applicable dans tous les cas, et dont la complexité est de l'ordre de n^3, où n est le nombre de nœuds. De plus, cet algorithme offre aussi le calcul de tous les chemins minimaux. Il s'agit en fait d'une généralisation au cas des graphes valués d'un algorithme de calcul de fermeture transitive d'un graphe.

Algorithme 1: Calcul des plus courts chemins

Roy-Warshall-Floyd (graph G,
double(node_value, node_value) L,
node_value(node_value, node_value) R
)
Données : Le graphe G
Résultat : $L(x, y)$: la table de valeur minimale des chemins de x à y.
Résultat : $R(x, y)$: la table de routage qui associe à chaque nœud y la valeur du successeur de x sur le meilleur chemin menant vers y.
On convient que si une valeur n'est pas présente alors alors $L(x, y) = +\infty$ et $R(x, y) = null$, c'est-à-dire qu'il n'y a pas de chemins de x à y.

```
Initialisation des tables (elles sont vidées de tout contenu)
```
$clear(L)$
$clear(R)$
```
Insertion des chemins de longueur un
```
for *all nodes* $x : get\_nodes(G)$ **do**
  for *all nodes* $y : get\_nodes(G, x)$ **do**
    $L(x, y) = longueur(get\_edges(G, x, y))$
    $R(x, y) = y$

```
Iteration sur la taille des chemins
```
for *all nodes* $z : get\_nodes(G)$ **do**
  for *all nodes* $x : get\_nodes(G)$ **do**
    for *all nodes* $y : get\_nodes(G)$ **do**
      $l = L(x, z) + L(z, y)$
      if $l < L(x, y)$ **then**
        $L(x, y) = l$
        $R(x, y) = z$

En calculant tous les plus courts chemins, cet algorithme détermine aussi les *composantes connexes* du graphe c'est à dire la partition du graphe en ensembles

de nœuds reliés entre eux par un chemin.

L'algorithme comporte donc trois boucles imbriquées, chacune de la taille du nombre de nœuds du graphe. A chaque étape de la boucle sur z les chemins minimaux de longueurs $1, 2, ...$ sont calculés. De fait, un chemin ne peut passer au pire que par n nœuds. Nous sommes donc assuré d'avoir calculé tous les chemins au bout de n étapes, ou plus exactement dès qu'une iteration de la boucle z n'apporte plus aucune amélioration. Il a été prouvé que, sur un graphe quelconque, cette complexité est optimale. Plusieurs améliorations existent, par exemple en tenant compte de propriétés particulières de certains graphes.

Bien entendu cet algorithme coûte non seulement de l'ordre de n^3 étapes de calcul mais aussi une structure de donnée dont la taille est de l'ordre de n^2. Si le graphe est très éparse, c'est-à-dire que beaucoup de nœuds n'ont pas de bords communs, les tables seront de taille réduite. C'est grâce à cette double structure de donnée $L(x, y)$ et $R(x, y)$ que nous pouvons mémoriser chaque étape de calcul et en propager le résultat à tous les nœuds du graphes, donc éviter l'explosion combinatoire d'un algorithme naïf. Selon l'application ces tables pourront être des tableaux indexés sur le numéro du nœud ou des tables associatives.

Cet algorithme bien que le plus élégant, n'est pas le plus asucieux ni le plus célèbre. Celui qui vient en tête est probablement l'algorithme dû à Edsger W. Dijkstra (1930-2002), qui est l'un des trois ou quatre informaticiens les plus importants du 20e siècle. Edsger W. Dijkstra reçut, en 1972, le prestigieux Turing Award, équivalent du Prix Nobel ou de la médaille Fields. Il publia son algorithme en 1959, l'année où il soutint sa thèse à l'Université d'Amsterdam. Cet algorithme a le même ordre de complexité et de consommation que celui présenté ici. Il nous fournit à chaque étape la valeur définitive d'un nœud. Ainsi, si on s'intéresse aux chemins de valeur minimale entre le nœud de départ et un nœud d'arrivée particulier, on peut arrêter le calcul dès que le nœud d'arrivée est calculé, même si tous les nœuds ne sont pas calculés. On dit que la stratégie est gloutonne, ce qui signifie que certains résultats ne seront plus remis en question (un glouton est quelqu'un qui avale et ne peut pas revenir en arrière, c'est-à-dire ne peut pas remettre en cause ce qu'il vient de faire). Pour l'algorithme de Roy-Warshall-Floyd, en revanche, aucun nœud ne peut être considéré comme calculé avant que tous les nœuds soient calculés : jusqu'à la fin de l'exécution (c'est-à-dire jusqu'à ce que la stabilité soit atteinte), toute valeur est susceptible d'être modifiée. Il s'agit là d'une stratégie de point fixe. Ce sont deux stratégies algorithmiques très générales.

Les grands algorithmes sur les graphes

Au delà du calcul du plus court chemin, de nombreux algorithmes sont à la fois de très grand intérêt théorique et pratique, regardons les principaux.

Cycle hamiltonien.

Un graphe hamiltonien est un graphe possédant au moins un cycle *hamiltonien*, c'est-à-dire un cycle passant par tous les nœuds une et une seule fois. Le problème dit du voyageur de commerce qui consiste à trouver le plus court chemin qui relie tous les nœuds est apparenté à ce problème. Dans les deux cas,

le problème est exponentiellement complexe, c'est-à-dire qu'il est impossible de le résoudre de manière exacte sans énumérer explicitement tous les chemins possibles (dont le nombre est exponentiellement grand) : c'est un problème NP-complet. En revanche, des méthodes approchées sont très efficaces. L'algorithme de Lin-Kernighan est l'une des meilleures heuristiques de ce type, elle consiste à échanger un nombre donné d'arêtes à partir d'une solution donnée pour trouver une solution de meilleur coût. À chaque étape la méthode décide du nombre d'arêtes à échanger pour trouver une tournée plus courte. Dans le problème de la tournée de véhicules, on généralise la problématique à plusieurs voyageurs, avec le but de minimiser le coût de livraison des biens. C'est un problème d'optimisation combinatoire et le domaine des sciences associées est dit de *recherche opérationnelle*. La gestion de stocks, la planification d'actions logistiques, l'optimisation de l'énergie ou du trafic sont autant d'exemples de grandes applications de ce domaine.

Recherche opérationnelle.

Une grande classe d'algorithme pour résoudre des problèmes de ce type de petite taille (jusqu'à une centaine de nœuds environ) est d'exprimer cela sous forme d'un problème de *programmation linéaire*. Le choix de passer par un nœud du graphe est alors modélisé sous forme de variables binaires $\mathbf{b} = (b_1, b_2, \cdots)^T$, ce qui permet alors :
- d'exprimer les contraintes topographiques du graphe sous forme d'inégalités linéaires de la forme $\mathbf{M}\,\mathbf{b} \geq \mathbf{n}$, pour une matrice $\mathbf{M}$ et un vecteur $\mathbf{n}$ donnés ;
- d'exprimer le critère de coût à optimiser sous forme d'une forme linéaire $\min \mathbf{c}^T\,\mathbf{b}$, pour un vecteur $\mathbf{c}$ donnés. L'optimisation linéaire est très souvent utilisée pour résoudre des problèmes combinatoires. Elle permet de résoudre très efficacement les problèmes dans lesquels les variables sont continues. L'algorithme du simplexe est le plus utilisé. Lorsqu'il y a des variables discrètes, optimisation linéaire et méthodes arborescentes peuvent être combinées.

L'*algorithme A** en est un exemple typique, c'est un algorithme de recherche de chemin dans un graphe entre un nœud initial et un nœud final, de taille telle que les algorithmes précédents ne sont pas utilisables (par exemple les combinaisons d'un jeu de plateau). Il utilise alors une évaluation heuristique sur chaque nœud pour estimer le meilleur chemin y passant, et visite ensuite les nœuds par ordre de cette évaluation heuristique. C'est un algorithme simple, ne nécessitant pas de prétraitement, et ne consommant que peu de mémoire. Il se combine avec d'autres algorithmes de recherche du plus court chemin.

Les méthodes *arborescentes par séparation et évaluation*, sont des méthodes génériques de résolution de problèmes d'optimisation combinatoire ou discrète. C'est une méthode d'énumération implicite : toutes les solutions possibles du problème peuvent être énumérées mais, l'analyse des propriétés du problème permet d'éviter l'énumération de larges classes de mauvaises solutions. La phase de séparation consiste à diviser le problème en un certain nombre de sous-problèmes de résoudre chaque sous-problème en évaluant la meilleure solution, puis de prendre globalement la meilleure solution trouvée. Ce principe de séparation peut être appliqué de manière récursive. Les ensembles de solutions (et leurs sous-problèmes associés) ainsi construits ont une hiérarchie naturelle en

arbre, souvent appelée arbre de recherche ou arbre de décision.

Colorisation d'un graphe.

Colorer un graphe signifie attribuer une couleur à chacun de ses nœuds de manière à ce que deux nœuds reliés par une arête soient de couleur différente. Est souvent recherché l'utilisation d'un nombre minimal de couleurs, dit *nombre chromatique*. Si le graphe est celui d'un réseau de communication coloriser est un problème d'allocation de fréquences de façon à ce que deux émetteurs (deux nœuds de ce graphe) géographiquement proches (donc connecté au niveau de ce graphe) ait bien des fréquences différentes qui n'interfèrent pas. C'est plus généralement le cas de l'allocation de ressources. Un tel problème reste largement ouvert et les algorithmes utilisés sont des heuristiques approximatives. Le théorème des quatre couleurs indique qu'il est possible, en n'utilisant que quatre couleurs différentes, de colorier n'importe quelle carte géographique découpée en régions connexes, de sorte que deux régions adjacentes ayant toutes une frontière en commun reçoivent toujours deux couleurs distinctes. C'est un très bel exemple de théorème démontré numériquement, par l'énumération informatique de tous les cas élémentaires possibles (soit 1478 cas critiques, pour plus de 1200 heures de calcul, en 1976), puis la preuve que le problème général se décompose en ces cas critiques. Le problème de la validation du théorème se trouve alors déplacé vers le problème de la validation de l'algorithme d'exploration, et de sa réalisation sous forme de programme. Il existe ainsi une version entièrement formalisée, formulée avec Coq par Georges Gonthier et Benjamin Werner, qui permet à un ordinateur de complètement vérifier le théorème des quatre couleurs.

Certains autres problèmes consistent à transformer ce graphe en un arbre (graphe sans cycle élémentaire) qui contient tous les nœuds d'une composante connexe du graphe et quelques arêtes. On parle alors d'un arbre couvrant du graphe. Cela sert à simplifier un câblage, optimiser un problème de trafic maritime ou aérien, etc..

Autres algorithmes.

D'autres algorithmes calculent des quantités sur un graphe soit
- en le parcourant en largeur, c'est-à-dire en listant les voisins d'un nœud pour ensuite les explorer un par un, soit
- en le parcourant en profondeur, c'est-à-dire à partir en explorant les chemins un par un, en marquant chaque nœud déjà parcouru.

Le parcours en largeur utilise une liste tampon dans laquelle on prend le premier nœud et place en dernier ses voisins non encore explorés. C'est une structure de donnée où le premier élément écrit dans la liste est le premier élément lu.
Le parcours en profondeur utilise une liste tampon dans laquelle on dépile un sommet et on empile ses voisins non encore explorés. C'est une structure de donnée où le dernier élément écrit dans la liste est le premier élément lu.

Structures arborescentes en informatique

La structure d'arbre avec un racine est omniprésente en informatique : système de fichier de l'ordinarteur, structure d'un document en chapitre et sous-

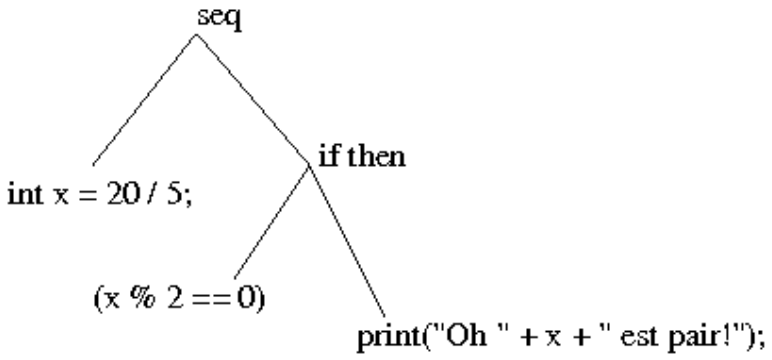
chapitre, etc.. Voyons d'autres exemples.

Un volume 3D se décompose en octree, avec à la racine la scène entière, puis de manière récursive, une partition de l'espace tridimensionnel à chaque niveau en huit octants, avec des applications immédiates à l'indexation spatiale et la détection efficace de collision entre des objets 3D.

Les arbres servent à définir des algorithmes de tri, de compression, etc..

C'est aussi une structure fondamentale pour la représentation d'un programme. Ainsi le mini programme suivant a t'il la structure arborescente correspondante :

```
int x = 20/5
if x%2 == 0 then
  print("Oh " + x + " est pair !")
```



Il peut donc être manipulé comme tout autre objet symbolique, pour être analysé, transformé, etc..