Algorithmique des graphes

Mines de Nancy
Pépites algorithmiques
Séances des 16 et 30 mars 2010

Gaëtan BISSON
gaetan.bisson@loria.fr

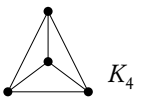
Inspiration

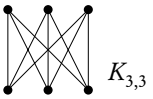
La notion de graphe formalise les structures que l'on qualifie communément de « réseaux » ; qu'il s'agisse d'un réseau téléphonique ou ferroviaire, un graphe n'est abstraitement qu'un ensemble de liens entre des nœuds.

Définition. *Un* graphe *est un couple d'ensembles* (S, A) *vérifiant* $A \subseteq S^2$ *; les éléments de S en sont les* sommets *et ceux de A les* arêtes*. Il est dit* non orienté *lorsque A est symétrique et irréflexif, ce que nous supposerons par la suite* [1].

On se convainc aisément du large champ d'application de cette notion ; citons-en quelques-uns des exemples concrets les plus évidents :

- circuit électrique : $S = \{\text{composants}\}$, $A = \{\text{fils}\}$
- réseau routier : $S = \{\text{échangeurs}\}$, $A = \{\text{routes}\}$
- mappemonde : $S = \{\text{pays}\}$, $A = \{\text{frontières}\}$





Parallèlement, nombreux sont les problèmes propres à cette structure de graphe (i.e. indépendants d'éventuels « réseaux » sous-jacents) ; parmi les plus classiques :

- Par où le courant passe-t-il entre deux composants ?
- Quelle est la plus courte route reliant deux villes données ?
- Comment colorier les pays de cette mappemonde ?

Nous ne pourrons aborder qu'une infime partie des problèmes et algorithmes qui composent ce vaste champ de recherche qu'est « l'algorithmique des graphes » ; le lecteur intéressé est donc vivement invité à compléter ce document par les ouvrages de référence mentionnés au début de chacune des sections.

N.B. Toutes les structures considérées ici seront sous-entendues finies.

1. À l'exception de la partie « bonus » 1.4.

1 Approche classique

Familiarisons-nous avec la notion de graphe en abordant quelques sujets classiques ; on complètera cela avec profit par l'excellent ouvrage [A].

1.1 Connexité

Une préoccupation première est de savoir si un réseau est d'un seul tenant.

Définition. *On appelle* voisins *d'un sommet x les sommets y vérifiant $(x,y) \in A$. La transitivisée de cette relation partitionne S en* composantes connexes.

Pour déterminer la composante connexe contenant un sommet w donné, on a recours à un *parcours en profondeur* qui consiste à explorer les sommets du graphe allant constamment le plus loin possible. Ceci s'implante par l'intermédiaire d'un ensemble X des sommets « déjà visités » et d'une pile Y des sommets « à visiter ».

Algorithme. COMPOSANTECONNEXE *(S,A,w)*
1. *Créer un ensemble $X = []$ et une liste $Y = [w]$.*
2. *Tant que Y n'est pas vide :*
3. *Enlever à Y son premier élément x et le rajouter à X.*
4. *Rajouter en tête de Y les voisins y de x non déjà dans $X \cup Y$.*
5. *Renvoyer X.*

Il est souvent utile d'étiqueter un graphe, c'est-à-dire d'apposer des informations sur certaines arêtes ou sommets (par exemple, les horaires de passage des trains en les gares d'un réseau ferroviaire). Le squelette de l'algorithme ci-dessus peut alors être utilisé afin de « propager » dans le graphe des modifications à apporter à ces étiquettes.

1.2 Chemins

Formalisons la notion de « déplacement » dans un graphe.

Définition. *Un* chemin *est une suite de sommets s_i reliés par des arêtes, ce qui s'écrit $(s_{i-1}, s_i) \in A$; on note cette chaîne d'arêtes*

$$s_0 \longrightarrow s_1 \longrightarrow s_2 \longrightarrow \cdots \longrightarrow s_n$$

et l'on dit que sa longueur *est l'entier n.*

La relation d'équivalence dictant quels sommets sont reliés par un chemin n'est d'autre que la transitivisée évoquée dans la définition précédente.

Un problème récurrent consiste à trouver le plus court chemin reliant deux sommets dans un graphe. Cela peut s'effectuer par un *parcours en largeur* qui consiste, à l'inverse du parcours en profondeur qui fonce tête baissée, à explorer en

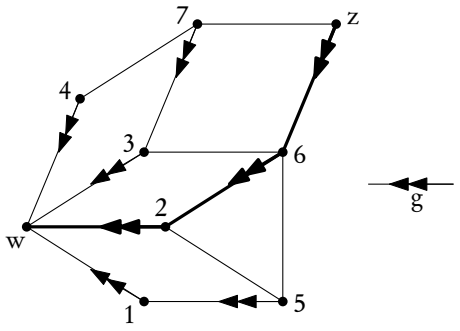


FIGURE 1 – Parcours en largeur pour la recherche du plus court chemin.

priorité les sommets proches ; Y est alors une file plutôt qu'une pile. [2] En outre, une fonction $g : S \to S$ décrira notre « provenance » dans le graphe, afin de pouvoir reconstruire le plus court chemin menant au point z donné.

Algorithme. PLUSCOURTCHEMIN *(S,A,w,z)*

1. *Créer un ensemble* $X = [\,]$ *et une liste* $Y = [w]$.
2. *Tant que* $z \notin Y$ *:*
3. *Enlever à* Y *son premier élément* x *et le rajouter à* X.
4. *Rajouter en queue de* Y *les voisins* y *de* x *non déjà dans* $X \cup Y$.
5. *Pour chacun de ces* y*, affecter* $g(y) = x$.
6. *Renvoyer le chemin* $z \to g(z) \to g(g(z)) \to \cdots$

La figure 1 illustre cet algorithme en indiquant l'ordre dans lequel les sommets x sont parcourus ainsi que la fonction g créée.

Exercice. Une fonction $f : A \to \mathbb{R}_+$ associe à chaque arête un poids ; adapter la méthode ci-dessus en l'algorithme de Dijkstra [4] qui trouve le chemin minimal en terme de somme des poids des arêtes ; discuter de sa complexité.

1.3 Structures de données

Il existe plusieurs façons de rendre effective la notion de graphe décrite plus haut, certaines plus adaptées à certains types de problèmes que d'autres. Pour mettre en avant la relation de voisinage, on stockera A sous forme d'une *liste d'adjacence*, c'est-à-dire d'un vecteur indicé par S dont la coordonnée en un sommet x contient ses voisins, en d'autres termes $y \in A_x \Leftrightarrow (x, y) \in A$.

Afin d'alléger notre travail, on fera usage de classes Java implantant des structures de données pertinentes pour notre problématique. Toute la documentation est en ligne :

`http://java.sun.com/javase/6/docs/api/`

2. En pratique, on utilisera toujours une liste mais en extraira les éléments en ordre inverse.

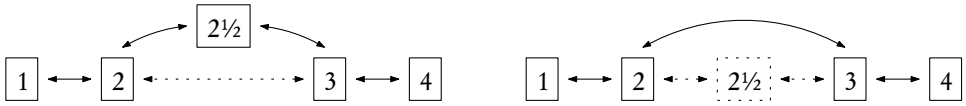


FIGURE 2 – Insertion et suppression dans une liste chaînée.

1.3.1 Listes

Une *liste* est une collection ordonnée d'objets destinée à être parcourue ; on veut aussi pouvoir y insérer et en enlever des éléments à des positions arbitraires. La structure de liste (doublement) chaînée stocke un élément en mémoire aux côtés de pointeurs vers les éléments suivant et précédant ; elle permet de réaliser en temps linéaire les opérations voulues (cf. figure 2) — et quitte à garder des pointeurs vers certains emplacements de la liste (e.g. début et fin) on peut y insérer et y enlever des éléments en temps constant.

En Java, le type `List<E>` est celui des listes formées d'objets de type `E`[3] ; la classe `LinkedList<E>` implante ce type sous forme de listes chaînées.

```
import java.util.* ;
List<String> X=new LinkedList<String>() ;
X.add("Quoi ?") ;
if (X.contains("Quoi ?")) X.add("Feur !") ;
for (String x  : X) System.out.println(x) ;
```

1.3.2 Ensembles et dictionnaires

Un *ensemble* est une collection d'objets « en vrac » sans répétition ; on désire y rajouter, en enlever et y rechercher des éléments. Une table de hachage stocke ces éléments dans un vecteur en plaçant l'élément x en position $h(x)$ où h est une (petite) fonction de hachage — on a donc une bonne équidistribution des éléments dans les cases du vecteur, ce qui permet de réaliser les opérations ci-dessus en temps amorti constant.

Une *fonction* assigne aux éléments d'un ensemble des valeurs que l'on veut pouvoir ajouter, supprimer et consulter ; ce n'est en somme qu'un ensemble dont chacun des éléments est muni d'un pointeur vers sa valeur.

En Java, la classe `HashSet<E>` réalise le type `Set<E>` des ensembles d'objets de type `E` ; pareillement, la classe `Hashtable<E,F>` réalise le type `Map<E,F>` des fonctions à valeurs de type `F`.

```
Map<String,Double> X=new Hashtable<String,Double>() ;
X.put("Nancy",7026.52) ; X.put("Metz",2966.98) ;
System.out.println(X.get("Nancy")) ;
```

3. Ce type doit être non primitif : on utilisera `Integer` au lieu de `int`.

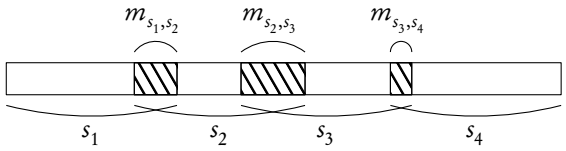


FIGURE 3 – Surmot et concaténation sans redondance.

1.4 Plus court surmot commun

Soit S un ensemble de mots (disons, sur l'alphabet usuel) ; le problème de trouver le plus petit mot contenant tous ceux de S intervient notamment en compression de données et en séquençage de génome.

Pour construire des surmots de S, concaténons ses éléments en éliminant la redondance qui apparaît entre mots successifs (cf. figure 3) : pour $(x,y) \in S^2$, soit $m_{x,y}$ le plus long suffixe de x qui est aussi préfixe de y ; notant $(s_i)_{i\in\{1,\dots,\#S\}}$ les mots de S dans l'ordre de concaténation, la longueur du surmot obtenu est

$$\sum_{s\in S} |s| - \sum_{i=1}^{\#S-1} |m_{s_i,s_{i+1}}|$$

où $|\cdot|$ dénote la longueur d'un mot. Le plus petit surmot provient donc de l'ordre de concaténation qui maximise la somme de droite.

Ainsi, il s'agit de trouver, dans le graphe $A = S^2$ muni de la fonction de poids $f : (x,y) \in A \mapsto |m_{x,y}|$, le chemin Hamiltonien (i.e. passant une unique fois par chaque sommet) le plus lourd. Ce problème étant NP-complet, on se contentera de chemins « très » longs ; on peut en trouver par un algorithme *glouton*, c'est-à-dire qui, à chaque étape, fait le choix qui semble localement le meilleur.

Algorithme. TRÈSLONGCHEMIN *(S,A)*

1. *Créer des ensembles $X = A$ et $Y = []$.*
2. *Tant que $X \neq \varnothing$:*
3. *Enlever à X son élément (x,y) maximisant $|m_{x,y}|$ et le rajouter à Y.*
4. *Retirer de X les éléments de la forme $(x,\cdot)$ ou $(\cdot,y)$.*
5. *Pour tout chemin $w \to \cdots \to z$ formé d'arêtes de Y, retirer (z,w) de X.*
6. *Renvoyer le chemin formé des arêtes de Y.*

L'étape 4 garantie qu'aucun sommet n'est visité deux fois et l'étape 5 assure que Y ne contient pas de cycle. Turner [6] a prouvé que la longueur du chemin trouvé est au moins la moitié de celle du chemin le plus long.

Fin de la première séance.

Ayez au minimum implanté l'algorithme PLUSCOURTCHEMIN.
Attaquez-vous ensuite au problème du plus petit surmot commun.

2 Approche algébrique

Voyons comment des outils algébriques s'emploient à l'étude des graphes ; ils nous permettront de considérer de façon « systématique » des familles de graphes ayant certaines propriétés. Pour pousser plus avant cette démarche, voir [B].

2.1 Matrice d'adjacence

Le pont reliant la théorie des graphes à l'algèbre est ceci.

Définition. *Soit (S,A) un graphe et fixons une bijection $S \simeq \{1,\dots,n\}$; sa* matrice d'adjacence *$A \in \mathrm{M}_n(\mathbb{Z})$ a 1 pour coefficient d'indice (x,y) si $(x,y) \in A$ et 0 sinon.*

Mettons en lumière la pertinence de cette représentation matricielle.

Lemme. *La puissance k^e de la matrice d'adjacence d'un graphe a pour coefficient d'indice (x,y) le nombre de chemins de longueur k reliant x à y.*

Corollaire. *Le nombre d'arêtes est $\frac{1}{2}\operatorname{tr}(A^2)$; le nombre de triangles est $\frac{1}{6}\operatorname{tr}(A^3)$.*

Évidemment, l'aspect matriciel ne s'arrête pas à ces quelques conséquences élémentaires de la multiplication. La matrice A étant symétrique réelle, elle est diagonalisable ; aussi semble-t-il opportun de considérer son spectre.

2.2 Spectre

Rappelons que le spectre d'une matrice est l'ensemble de ses valeurs propres et esquissons son intérêt pour les graphes par deux résultats sans grande prétention.

Proposition. *Dans un graphe connexe dont la matrice d'adjacence a k valeurs propres distinctes, on peut relier tout couple de sommets en au plus $k-1$ arêtes.*

Démonstration. Par le lemme ci-dessus, s'il existe un plus court chemin de longueur k, les matrices A^i sont linéairement indépendantes pour $i \in \{0,\dots,k\}$; or, la dimension de l'espace qu'elles engendrent est bornée par le degré du polynôme minimal de A, qui n'est autre que son nombre de valeurs propres distinctes. □

Le problème du coloriage consiste à assigner une couleur à chaque sommet de sorte qu'aucun n'ait la même couleur que l'un de ses voisins. Un algorithme simple de coloriage donne la borne suivante, où μ_1 désigne la plus grande valeur propre (et dorénavant μ_2 désignera la suivante).

Proposition. *Un graphe peut se colorier en $1+\mu_1$ couleurs.*

Démonstration. Pour $i \in \{1,\dots,n\}$, définissons s_i comme le sommet de plus petit degré dans le graphe privé des sommets $\{s_j : j < i\}$ et des arêtes qui en partent ; ces degrés sont inférieurs à μ_1. Colorier alors les s_i en ordre inverse, choisissant pour chacun une couleur non déjà assignée à un de ses voisins. □

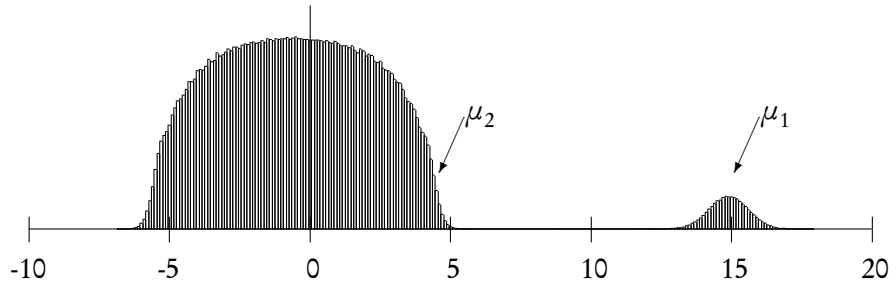


FIGURE 4 – Distribution spectrale moyenne sur 10^5 graphes ($\#S = 30$, $\alpha = 1/2$).

En réalité, le spectre est intimement lié à de profondes propriétés du graphe que nous aborderons plus tard en nous restreignant au cadre ci-décrit.

Définition. *Le* degré *d'un sommet est le nombre d'arêtes qui en partent. On qualifie de* d*-*régulier *les graphes dont chacun des sommets a pour degré* d*.*

La théorème suivant, à la démonstration laissée en exercice, est fondamental.

Théorème. *Les valeurs propres d'un graphe* d*-régulier sont inclues dans l'intervalle réel* $[-d;d]$*; elle-même,* d *a pour multiplicité le nombre de composantes connexes.*

2.3 Algorithmes

Attelons-nous à la description d'une méthode permettant de calculer le spectre d'une matrice symétrique réelle A donnée. Nous procéderons en deux étapes :

1. Calculer le polynôme caractéristique de A.
2. En déterminer les racines.

Applications. Observer, pour $\alpha \in]0;1[$, les spectres de graphes construits en donnant à chaque arête de S^2 une probabilité α d'appartenir à A (figure 4).

En fait, on prouve que la densité spectrale de graphes connexes d-réguliers converge, à normalisation près, vers $\sqrt{4(d-1)-x^2}/(d^2-x^2)\,\mathrm{d}x$, la longueur du plus petit cycle tendant vers l'infini. Construire de telles familles (e.g. $d = 2$).

2.3.1 Polynôme caractéristique

Calculons le polynôme caractéristique de A par sa définition pragmatique comme déterminant de la matrice $A - x\,\mathrm{I}$ de $\mathrm{M}_n(\mathbb{Z}[x])$ via la méthode du pivot de Gauss. Elle consiste à la mettre sous forme trigonale par une succession d'opérations élémentaires sur les lignes; ces dernières correspondent à des multiplications par des matrices de transvection et n'affectent donc pas le déterminant.

Notons m_{ij} le coefficient d'indice (i,j) de la matrice, et L_i sa i^{e} ligne. L'opération typique, permettant d'annuler m_{ji}, est $L_j \leftarrow L_j - m_{ji}/m_{ii}L_i$. Afin de ne pas introduire de fractions, on peut multiplier les L_j par les m_{ii}, quitte à penser plus tard à diviser par ces quantités le déterminant calculé.

Lewis Carroll [2] a montré comment ce faire en évitant que les coefficients ne grossissent trop : il s'agit de remarquer que le quotient de l'étape 3 est entier. C'est ce qu'on connaît aujourd'hui sous le nom de « méthode de Gauss–Bareiss ».

Algorithme. DÉTERMINANT *(M)*

1. *Pour i de* 1 *à n :*
2. *Pour j de $i+1$ à n :*
3. *Poser* $L_j \leftarrow (m_{ii}L_j - m_{ji}L_i)/m_{(i-1)(i-1)}$.
4. *Renvoyer* m_{nn}.

On aura posé, par convention, $m_{00} = 1$.

2.3.2 Racines réelles

Pour localiser les racines du polynôme P obtenu, appuyons-nous sur un résultat de Sturm [1]. Il concerne les polynômes sans facteurs carrés, cas auquel on se ramènera en mettant de côté les racines multiples données par $\operatorname{pgcd}(P, P')$.

Théorème. *Soit P un polynôme de $\mathbb{R}[x]$ sans facteur carré ; posons*

$$P_0 = P \qquad P_1 = P' \qquad P_{n+1} = -\operatorname{reste}(P_{n-1}, P_n)$$

et notons $\sigma(x)$ le nombre de changements de signe de la suite $P_i(x)$. On peut alors exprimer le nombre de racines de P dans tout intervalle $]\alpha,\beta] \subseteq \mathbb{R}$ par $\sigma(\alpha) - \sigma(\beta)$.

Pour récupérer des valeurs numériques de ces racines, il ne reste qu'à appliquer le principe de dichotomie ; cela consiste à découper récursivement l'intervalle de recherche $]\alpha,\beta]$, s'arrêtant lorsque la précision voulue ε est atteinte.

Algorithme. RACINESRÉELLES *(P,α,β)*

1. *Poser* $\delta = \frac{\alpha+\beta}{2}$.
2. *Si P a des racines sur* $]\alpha,\beta]$ *:*
3. *Si* $\beta - \alpha \leqslant \varepsilon$ *: afficher* δ.
4. *Sinon :*
5. *Lancer* RACINESRÉELLES*(P,α,δ)*.
6. *Lancer* RACINESRÉELLES*(P,δ,β)*.

Fin de la seconde séance.

Ayez implanté le calcul du spectre d'ici la fin de la séance prochaine.
On s'aidera de la bibliothèque de polynômes fournie en annexe.

3 Graphes expanseurs

Pour des éclaircissements sur le contenu de cette section, consulter [C].

3.1 Motivation et quantification

La théorie des graphes trouve une application phare en l'étude des réseaux de transport et de communication dont une des propriétés clefs est la résistance aux coupures. C'est précisément ce que mesure l'indicateur suivant.

Définition. *La constante d'expansion d'un graphe (S,A), notée h, vaut*

$$\min\left\{\frac{|\partial T|}{\min\{|T|,|S\setminus T|\}} : T\subseteq S\right\}$$

où $\partial T = (T\times S\setminus T)\cap A$ contient les arêtes reliant T au reste du graphe.

Ainsi, pour la construction de réseaux redondants, on recherche des graphes ayant de grandes constantes d'expansion ; toute liaison ayant un coût, on souhaite aussi en limiter le nombre d'arêtes. Fixons donc un entier d et restreignons notre étude aux graphes connexes d-réguliers.

Théorème. *La constante d'expansion d'un graphe connexe d-régulier est encadrée de la sorte :*

$$\frac{\mu_1-\mu_2}{2}\leqslant h\leqslant\sqrt{2\mu_1(\mu_1-\mu_2)}$$

La grandeur $\mu_1-\mu_2$, mise en lumière par la figure 4, joue un rôle primordial. Or $\mu_1 = d$ et maximiser h revient alors à minimiser μ_2 ; toutefois, cela n'est pas possible à outrance, μ_2 étant bornée asymptotiquement.

Théorème. *Toute ensemble infini de graphes connexes d-réguliers vérifie*

$$\liminf\mu_2\geqslant 2\sqrt{d-1}.$$

Un graphe connexe d-régulier vérifiant $\mu_2\leqslant 2\sqrt{d-1}$ est dit *de Ramanujan*. Décrivons maintenant la construction d'une famille infinie de tels graphes [5].

3.2 Graphes de Cayley

Continuons d'exploiter les bienfaits de l'algèbre.

Définition. *Soit H une partie symétrique (i.e. close pour l'inversion) d'un groupe G ; on appelle graphe de Cayley de H le graphe $S = G$ et $A = \{(x,yx) : x\in G, y\in H\}$.*

Le graphe de Cayley de $\{\pm 1\}\subset\mathbb{Z}/n$ est donc naturellement le cercle à n sommets. On peut construire ainsi profusion de graphes dont les propriétés peuvent être étudiées algébriquement ; que dire, par exemple, de $\{2^{\pm 1},3^{\pm 1}\}\subset\mathbb{F}_p^\times$?

3.3 Construction de Lubotzky–Phillips–Sarnak

Afin de raccourcir notre présentation, nous prendrons quelques hypothèses supplémentaires par rapport à la construction générale.

Étape 1. Fixer un premier p congru à 1 mod 4. Énumérer l'ensemble $\mathscr{S}_p$ des quadruplets d'entiers (w,x,y,z) avec w impair positif vérifiant l'équation

$$w^2+x^2+y^2+z^2=p.$$

Étape 2. Fixer un grand premier q congru à 1 mod 4 et modulo lequel p est un carré. Calculer une racine carrée i de -1 mod q. Noter φ le morphisme :

$$(w,x,y,z)\in\mathbb{N}^4\mapsto\begin{pmatrix} w+ix & y+iz \\ -y+iz & w-ix \end{pmatrix}\in \mathrm{M}_2(\mathbb{F}_q)$$

Étape 3. Construire $\mathscr{X}_p^q$, le graphe de Cayley de $\frac{1}{\sqrt{p}}\varphi(\mathscr{S}_p)\subset \mathrm{PSL}_2(\mathbb{F}_q)$.

Rappelons que le groupe spécial linéaire PSL_2 est celui des matrices carrées de taille deux et de déterminant unité où l'on identifie une matrice à son opposée.

Théorème. *Le graphe $\mathscr{X}_p^q$ est connexe, $(p+1)$-régulier et de Ramanujan.*

3.4 Arithmétique effective

Idéalement, on utiliserait une bibliothèque multiprécision afin de manipuler des entiers de taille arbitraire ; en Java, la classe `math.BigInteger` implante des algorithmes adaptés à des entiers de taille moyenne. [4] En pratique, on se contentera largement de graphes $\mathscr{X}_p^q$ dont le cardinal, $\#\mathrm{PSL}_2(\mathbb{F}_q)=q(q^2-1)/2$, est dans l'intervalle $[-2^{63};2^{63}-1]$ qu'est celui des entiers de type `long` ; on calculera donc dans $\mathbb{F}_q$ en réduisant modulo q les résultats des opérations de type `long`.

Pour finir, expliquons comment effectuer ces opérations intrinsèques à $\mathbb{F}_q$ que sont l'inversion et l'extraction de racines carrées.

La méthode Java `math.BigInteger.modInverse` implante l'algorithme d'Euclide étendu ; il consiste à calculer le PGCD de x et q, exploitant les résultats intermédiaires afin de construire une relation de Bézout. Explicitement, posons :

$$\begin{matrix} x_0=q, x_1=x \\ u_0=0, u_1=1 \end{matrix} \quad\text{et}\quad \begin{matrix} x_{i+1}=x_i-q_ix_{i-1} \\ u_{i+1}=u_i-q_iu_{i-1} \end{matrix} \quad\text{où}\quad q_i=\lfloor x_i/x_{i-1}\rfloor$$

La suite x_i est décroissante et son plus petit terme non nul x_n est le PGCD de x et q, c'est-à-dire que $x_n=1$ puisque q est premier et $x\neq 0$ mod q. La suite u_i vérifie $u_ix=x_i$ mod q pour tout i ; le terme u_n est donc l'inverse de x modulo q.

4. La bibliothèque de polynômes devrait elle-même utiliser la classe `math.BigDecimal` en lieu du type `double`, choisi à fins de simplicité. On pourra d'ailleurs la voir atteindre ses limites.

Reste la méthode de Tonelli [3] pour l'extraction de racines carrées dans $\mathbb{F}_q$. Notons $q-1 = 2^s t$ avec t impair, de sorte que $\mathbb{F}_q^\times$ soit isomorphe à $\mathbb{Z}/2^s \times \mathbb{Z}/t$. On voit ainsi qu'un élément x est un carré si et seulement si $x^{(q-1)/2} = 1$, soit encore si et seulement si x^t est un carré. Cela montre que les propriétés quadratiques se jouent exclusivement dans le premier facteur ; il correspond au sous-groupe S_{2^s}, où l'on a noté $S_n = \{x \in \mathbb{F}_q^\times : x^n = 1\}$.

Projetons donc x en $x^t \in S_{2^s}$. Exprimant x^t comme puissance d'un générateur z de S_{2^s}, disons $x^t = z^k$, on voit que x est un carré si et seulement si k est pair, auquel cas $\sqrt{x^t} = z^{k/2}$ d'où l'on déduit $\sqrt{x} = x^{(t+1)/2} z^{-k/2}$.

Algorithme. RACINECARRÉ *(x,q)*

1. *Écrire* $q-1$ *sous la forme* $2^s t$ *avec* t *impair ; poser* $k = 0$.
2. *Calculer* z, *la puissance* t^e *d'un non carré quelconque de* $\mathbb{F}_q^\times$.
3. *Pour* i *de* 0 *à* $s-1$ *:*
4. *Si* $(x^t/z^k)^{2^{s-i-1}} \neq 1$, *ajouter* 2^i *à* k.
5. *Renvoyer* $x^{(t+1)/2} z^{-k/2}$.

L'étape 4 détermine la i^{e} décimale de l'écriture binaire de k, caractérisée par la propriété $(x^t/z^k) \in S_{2^{s-i-1}}$; c'est-à-dire que l'on approche progressivement x^t par des puissances de z dans les sous-groupes successifs $S_{2^1} \subset S_{2^2} \subset$ etc.

Fin de la dernière séance.

Ayez au minimum implanté le calcul du spectre.
Attaquez-vous ensuite à la construction de graphes expanseurs.

Articles de recherche

[1] Jacques Charles François Sturm. Mémoire sur la résolution des équations numériques. *Bulletin des sciences de Férussac*, volume 11, page 419–422, 1829.

[2] Charles Lutwidge Dodgson. Condensation of determinants, being a new and brief method for computing their arithmetical values. *Proceedings of the Royal Society of London*, volume 15, pages 150–155, 1866.

[3] Alberto Tonelli. Bemerkung über die Auflösung quadratischer Congruenzen. *Göttinger Nachrichten*, pages 344–346, 1891.

[4] Edsger Wybe Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, volume 1, pages 269–271, 1959.

[5] Alexander Lubotzky, Ralph Phillips et Peter Sarnak. Ramanujan graphs. *Combinatorica*, volume 8, numéro 3, pages 261–277, 1988.

[6] Jonathan Shields Turner. Approximation algorithms for the shortest common superstring problem. *Information and Computation*, volume 83, numéro 1, pages 1–20, 1989.

Ouvrages de référence

[A] Sanjoy Dasgupta, Christos Papadimitriou et Umesh Vazirani. *Algorithms*. McGraw-Hill, 2006.

[B] Chris Godsil et Gordon Royle. *Algebraic Graph Theory*. Springer-Verlag, 2001.

[C] Giuliana Davidoff, Peter Sarnak et Alain Valette. *Elementary number theory, group theory, and Ramanujan graphs*. Cambridge University Press, 2003.

Bibliothèque de polynômes

Cette classe Java implante les algorithmes triviaux de manipulation de polynômes ; son efficacité est raisonnable en petit degré. Ne pas utiliser aveuglément !

```
class Polynom {
    double[] C ; int d ;

    int deg() {
        int i=C.length-1 ;
        while (i>=0 && Math.abs(C[i])<1.E-10) i-- ;
        return i ;
    }

    Polynom(double[] T) { C=T ; d=deg() ; }
    Polynom(double t,int u)
        { double[] T=new double[1+u] ; T[u]=t ; C=T ; d=u ; }

    static Polynom Zero=new Polynom(0,0) ;
    static Polynom One=new Polynom(1,0) ;
    static Polynom X=new Polynom(1,1) ;

    Polynom plus(Polynom P) {
        double[] T=new double[1+Math.max(P.d,d)] ;
        for (int i=0 ;i<=d ;i++) T[i]+=C[i] ;
        for (int i=0 ;i<=P.d ;i++) T[i]+=P.C[i] ;
        return new Polynom(T) ;
    }

    Polynom minus(Polynom P) {
        double[] T=new double[1+Math.max(P.d,d)] ;
        for (int i=0 ;i<=d ;i++) T[i]+=C[i] ;
        for (int i=0 ;i<=P.d ;i++) T[i]-=P.C[i] ;
        return new Polynom(T) ;
    }
```

```
    Polynom times(Polynom P) {
        double[] T=new double[2+P.d+d] ;
        for (int i=0 ;i<=d ;i++)
            for (int j=0 ;j<=P.d ;j++)
                T[i+j]+=C[i]*P.C[j] ;
        return new Polynom(T) ;
    }

    Polynom[] remquo(Polynom P) {
        double[] T=new double[1+d] ;
        for (int i=0 ;i<=d ;i++) T[i]=C[i] ;
        Polynom R=new Polynom(T),Q=Polynom.Zero,L ;
        while (R.d>=P.d) {
            L=new Polynom(R.C[R.d]/P.C[P.d],R.d-P.d) ;
            Q=Q.plus(L) ;
            R=R.minus(P.times(L)) ;
        }
        Polynom[] RQ={R,Q} ;
        return RQ ;
    }

    Polynom gcd(Polynom P) {
        Polynom R=this,S=P,T ;
        while (S.d !=-1) { T=S ; S=R.remquo(S)[0] ; R=T ; }
        return R ;
    }

    Polynom deriv() {
        double[] T=new double[d] ;
        for (int i=1 ;i<=d ;i++) T[i-1]=i*C[i] ;
        return new Polynom(T) ;
    }

    double eval(double x) {
        double Px=0 ;
        for (int i=d ;i>=0 ;i--) { Px*=x ; Px+=C[i] ; }
        return Px ;
    }

}
```