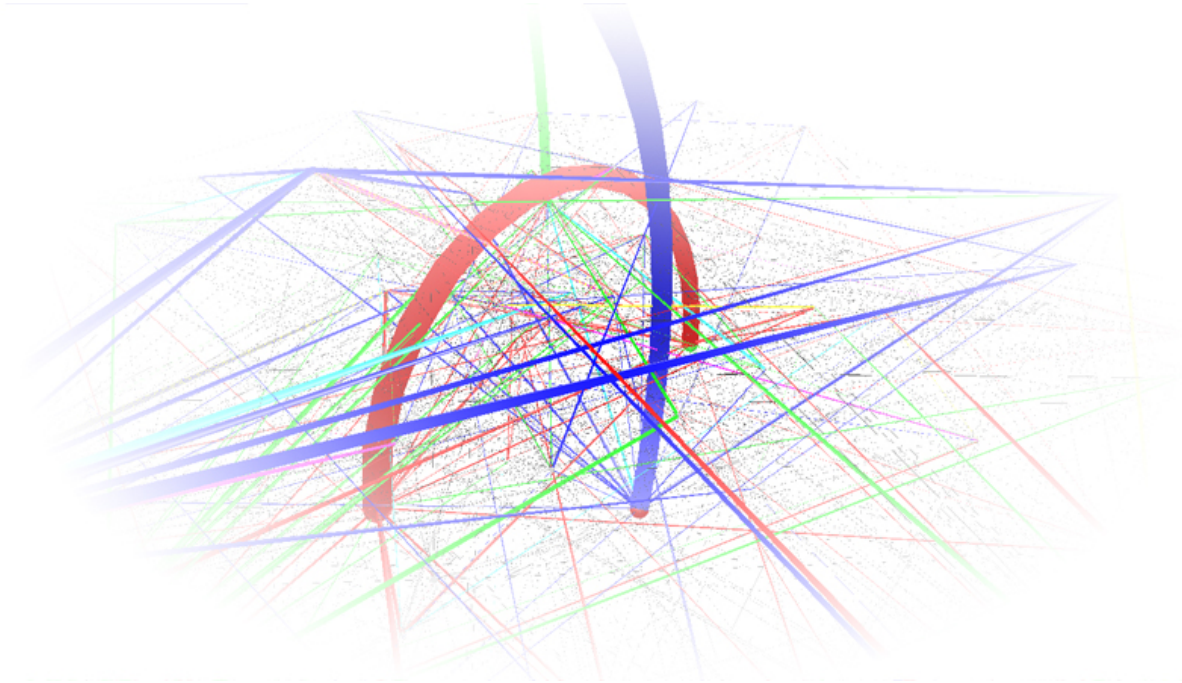
Plongements de graphes et distorsion



Michaël Monerau & Arnaud de Mesmay

Juin 2008

Sous la direction de Pierre Pansu & Éric Colin de Verdière

Plan

Présentation

L'objet de ce mémoire est de présenter des résultats récents dans le domaine des mathématiques combinatoires apportés par une nouvelle façon de considérer les graphes : leur point de vue géométrique. Les graphes peuvent en effet être munis de métriques reflétant en partie ou en totalité leurs propriétés. La plus évidente est par exemple la métrique du plus court chemin, qui à deux points u et v associe la taille du plus court chemin allant de u à v. Mais ces métriques n'ont généralement rien à voir avec celles que l'on manipule usuellement : ce ne sont pour la plupart que des semi-métriques (la distance entre deux points distincts peut être nulle), et pour les étudier, on aimerait les comparer avec les métriques que l'on maîtrise bien, c'est-à-dire celles des espaces ℓ_p (on désignera comme tels tous les $\mathbb{R}^k$ munis de la distance $d(x,y) = (\sum_{i=1}^{k} |x_i - y_i|^p)^{\frac{1}{p}}$). Le cas optimal est celui où l'on peut plonger le graphe muni de sa métrique isométriquement dans un tel espace, c'est par exemple le cas pour tout graphe fini de cardinal n dans l'espace ℓ_∞ : il suffit d'associer à chaque point x le vecteur $(d(x,y))_{y\in G}$. Malheureusement, ce n'est pas le cas pour les espaces ℓ_p pour p fini, comme le montre l'exemple du graphe $K_{1,3}$ (dessin *a.*). On ne peut pas le plonger dans ℓ_2 puisque le sommet central devrait être au milieu des segments reliant les trois autres points entre eux alors que ces trois points sont distincts. On peut par contre le plonger dans ℓ_2 avec une certaine distorsion(dessin *b.*), c'est-à-dire en autorisant les distances à se déformer un peu lors du passage d'un espace à l'autre. À l'aide de ces plongements "presque isométriques", on obtient ainsi des solutions élégantes à quelques problèmes combinatoires classiques.

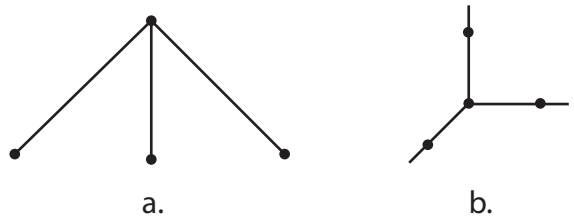


FIG. 1 – Exemple de graphe non plongeable dans $\mathbb{R}^n$

Les travaux de Linial, London et Rabinovich, notamment [LLR95], permettent pour la première fois d'appliquer ces plongements avec distorsion pour proposer des algorithmes de résolution approchés de deux problèmes combinatoires fondamentaux : SPARSEST CUT et MULTICOMMODITY FLOW. Ils utilisent un résultat de plongement initialement énoncé par Bourgain [Bou85]. Cette approche géométrique permet de développer un algorithme polynomial d'approximation de SPARSEST CUT en $O(\log n)$, ce qui était, à la date de rédaction de l'article, le meilleur résultat obtenu. En remarquant qu'on n'a pas besoin de plonger une métrique quelconque dans ℓ_1 mais seulement les métriques de type négatif (voir 2.2.2 pour la définition précise) pour approximer correctement SPARSEST CUT, les travaux d'Arora, Lee et Naor ainsi que de Rao et Vazirany ont permis ensuite d'améliorer cet algorithme d'approximation tout en utilisant les mêmes techniques.

Dans un premier temps, nous définissons les différentes notions employées (semi-métriques, graphes, coupes, plongements lipschitziens) et nous énonçons les principaux résultats qui y sont liés. Dans la deuxième partie, nous appliquons ces techniques de plongement pour détailler le théorème de Linial, London et Rabinovich concernant les MULTICOMMODITY FLOWS. Dans la troisième partie, nous verrons les améliorations successives apportées à cet algorithme permettant dans des cas particuliers d'améliorer l'algorithme pour une approximation en $O(\sqrt{\log n}\log\log n)$ [ALN08], voire $O(\sqrt{\log n})$ dans le cas de demandes uniformes [ARV04].

Nous tenons à remercier Pierre Pansu et Éric Colin de Verdière pour leur aide, leur disponibilité, leur tenacité et leur pugnacité tout au long de ce travail avec nous.

1 Complexité et algorithmique

1.1 Complexité en temps

Pour quantifier la rapidité d'exécution d'un algorithme, il est essentiel de disposer d'outils logiques et mathématiques puissants mesurant leur *complexité*, c'est-à-dire le nombre d'opérations qu'ils vont effectuer au cours d'une exécution typique.

Un algorithme fonctionne sur des données d'entrée : l'instance particulière du problème qu'il doit résoudre. On aimerait pouvoir quantifier le temps d'exécution de l'algorithme en fonction de la taille de l'entrée qu'on lui donne. Il est essentiel de définir précisément quelles sont les entrées, et quelles sont leur taille. Par exemple, pour un algorithme prenant en entrée un graphe, on peut considérer que la taille des entrées est le nombre de sommets plus le nombre d'arêtes.

On dit alors qu'un problème est de complexité f (où f est une fonction réelle) si lorsqu'on lui fournit une entrée de taille n, il retourne sa réponse en temps $O(f(n))$ lorsque n tend vers l'infini.

1.2 Classes de complexité

On regroupe les différents problèmes algorithmiques en plusieurs classes. Tout d'abord, on se restreint aux problèmes de décision : ceux auxquels on répond par *oui* ou *non*.

On dit qu'un problème de décision est dans **P** s'il se résout à l'aide d'un algorithme *déterministe* qui s'exécute en *temps polynomial* en la taille des données d'entrée.

Un problème de décision est **NP** s'il existe un algorithme *non déterministe* en *temps polynomial* qui le résout. Donc $\mathbf{P} \subseteq \mathbf{NP}$.

Les problèmes **NP** ne sont pas tous équivalents entre eux, mais on peut trouver des problèmes, dits **NP-Complets**, qui, en quelque sorte, capturent toute la complexité de la classe **NP** : si on savait les résoudre en temps polynomial déterministe, on saurait résoudre tous les autres **NP** en temps polynomial déterministe. À l'heure qu'il est, aucun problème de **NP-Complet** n'a reçu de solution polynomiale déterministe. Une des grandes conjectures, unanimement reconnue, est en effet que $P \subsetneq NP$.

On peut voir les problèmes **NP-Complets** comme des problèmes qu'on n'a pas espoir de résoudre directement, au moins sur les ordinateurs classiques. Les seuls algorithmes les résolvant sont typiquement de complexité exponentielle, ce qui interdit toute résolution automatisée.

On peut alors étendre naturellement cette notion aux problèmes qui ne sont pas de décision (par exemple où il faut expliciter une solution et pas seulement répondre par *oui* ou *non*) : on dit qu'un problème est **NP-Difficile** s'il est au moins aussi difficile que les **NP-Complets**. Autrement dit, si on connaît un algorithme déterministe résolvant un problème **NP-Difficile**, alors on pourrait résoudre tous les problèmes **NP-Complets** avec la même complexité, et donc aussi tous les problèmes **NP**. Les problèmes **NP-Difficiles** doivent donc être vus comme des questions très complexes à répondre. On peut même affirmer qu'il est impossible d'y répondre actuellement en moins de quelques centaines d'années sur des entrées quelconques.

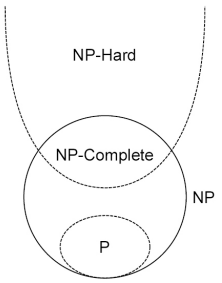


FIG. 2 – Relation entre les différents ensembles de complexité (si $P \subsetneq NP$)

1.3 Nos problèmes

On trouve dans [DL97] une liste des complexités des problèmes qui nous intéressent. Pour la définition des problèmes, on se reporte à la suite. Les complexités sont introduites ici pour motiver notre étude.

Max-cut (décision)	NP-complet
Max-cut (optimisation)	NP-difficile
Max-flow (optimisation)	P
Sparsest-cut (optimisation)	NP-difficile
Cut cone (appartenance + séparation)	NP-difficile
Cut cone (appartenance)	NP-complet

Notre objectif est d'apporter une réponse au problème Sparsest-cut. Celui-ci étant **NP-Complet**, on s'intéresse à en trouver une solution approchée. Notre périple va nous conduire à munir le graphe concerné d'une semi-métrique adéquate pour pouvoir le plonger dans des espaces ℓ_p en lui appliquant des résultats topologiques bien connus, qu'on pourrait croire indépendants *a priori* du monde algorithmique. Ensuite, on pourra manipuler ce nouvel objet de manière à résoudre le problème initial d'une manière à la fois originale et élégante.

Pour cela, on va d'ailleurs utiliser le problème Max-flow qui, lui, est résolvable en temps polynomial déterministe.

2 Semi-métriques et distorsion

Comme annoncé en introduction, le cœur de notre étude consiste à plonger des graphes[1] dans des espaces ℓ_p. Il convient donc de commencer notre travail par l'exploration des quelques résultats de ce domaine que nous utiliserons.

Nous allons définir quelques termes de géométrie discrète, puis caractériser les semi-métriques ℓ_p-*plongeables* d'abord strictement puis en s'autorisant une petite marge de manœuvre.

2.1 Géométrie discrète et semi-métriques

2.1.1 Polytopes et polyèdres

Les ensembles que l'on rencontre tout au long de l'étude ont très souvent une structure qui se comprend bien en termes géométriques car ils sont inclus dans $\mathbb{R}^n$ et issus d'un nombre fini d'équations ou inéquations linéaires. Posons quelques définitions pour illustrer cela :

[2.A] Définition ***(Polytope)***
Un polytope est l'enveloppe convexe d'un nombre fini de points de $\mathbb{R}^n$.

[2.B] Définition ***(Polyèdre)***
Un polyèdre est un ensemble de la forme :

$$\{x \in \mathbb{R}^n \quad | \quad Ax \leq b\}$$

où A est une matrice $m \times n$ et b un vecteur de $\mathbb{R}^m$, pour un certain $m \geq 0$.

Comme on s'y attend, ces deux définitions se rejoignent grâce au :

[2.C] Théorème ***(Minkowski-Weyl)***
Un polytope est un polyèdre borné et réciproquement.

De même, on peut donner un vocabulaire analogue pour les cônes :

[2.D] Définition ***(Cône convexe)***
Un cône convexe est un ensemble de la forme $\mathbb{R}_+(X) = \{\sum_{i=1}^n \lambda_i x_i, n \geq 0, \lambda_i \in \mathbb{R}_+,\ x_i \in X\}$. Il est de type fini s'il existe un tel X fini.

[2.E] Définition ***(Cône polyédral)***
Un cône polyédral est un cône défini par un nombre fini d'équations :

$$C = \{x \in \mathbb{R}^n \quad | \quad Ax \leq 0\}$$

où A est une matrice $m \times n$, pour un certain $m \geq 0$.

Heureusement, on a ici aussi le :

[2.F] Théorème ***(Minkowski-Weyl 2)***
Un cône convexe de type fini est un cône polyédral et réciproquement.

[1] Tous les graphes que nous considérerons seront supposés connexes, non orientés, sans arête multiple et sans boucle.

2.1.2 Semi-métriques

Nous allons avoir besoin, pour quantifier nos manipulations sur les graphes, de définir des métriques sur différents espaces. Mais nous utiliserons des semi-métriques plutôt que des métriques.

Rappelons qu'une semi-métrique est une métrique où l'on n'impose pas que la distance entre deux points distincts soit non nulle. Une semi-métrique sur un espace $V_n = \{1, \dots, n\}$ à n élements peut être représentée par un vecteur de $\mathbb{R}^{E_n} \simeq \mathbb{R}^{\frac{n(n-1)}{2}}$, satisfaisant donc l'inégalité triangulaire, où $E_n = \{(i,j) \in [\![1,n]\!]^2 \mid i < j\}$.

En particulier, on peut poser la :

[2.G] Définition ***(Semi-métrique de coupe)***

Soit S une coupe de V_n, c'est-à-dire un sous-ensemble de V_n. La semi-métrique de coupe associée à S, notée $\delta(S)$, est :

$$\delta(S)_{i,j} = \begin{cases} 1 & si\ (i \in S\ et\ j \notin S) \\ 1 & si\ (i \notin S\ et\ j \in S) \\ 0 & sinon \end{cases}$$

Ces semi-métriques de coupe joueront un rôle très particulier dans la suite, et c'est pourquoi elles méritent qu'on distingue la :

[2.H] Définition ***(Cut cone)***

On définit Cut_n comme le cône convexe de $\mathbb{R}^{E_n}$ engendré par les semi-métriques de coupe.

Malgré son apparente simplicité, Cut_n est un objet combinatoire encore très mystérieux. En effet, on ne connaît pas, à l'heure actuelle, toutes les équations définissant les "facettes" de Cut_n, sauf si $n \leq 7$ ([DL97], p. 50).

Dans le même ordre d'idées, on peut montrer que le problème de décider si une métrique appartient ou non à Cut_n est **NP-complet** [DL97, p.49]. C'est d'ailleurs précisément ce caractère NP-complet qui va nous pousser, par la suite, à restreindre nos objectifs de résultats pour obtenir un problème plus abordable en terme de résolution effective.

2.1.3 Le cas des graphes

Un espace métrique fini $V_n = \{1 \dots n\}$, de métrique d, peut être vu comme un graphe complet non-orienté où la distance d est interprétée comme un poids sur les arêtes.

Une semi-métrique n'est alors rien d'autre qu'une fonction de poids sur les arêtes (respectant l'inégalité triangulaire).

Mentionnons d'ores et déjà une métrique qui nous sera utile par la suite : sur un graphe complet $G = (V, E)$, on définit la *métrique de distance minimum* sur G comme :

$$\delta_G(i,j) \text{ est le nombre d'arêtes minimum sur un chemin reliant } i \text{ à } j.$$

2.2 Plongements ℓ_p

Commençons par rappeler la :

[2.I] Définition ***(Plongement isométrique)***

Soient (X,d) et (X',d') deux espaces semi-métriques. On dit que (X,d) se plonge isométriquement dans (X',d') s'il existe une fonction $f : X \longrightarrow X'$ telle que :

$$\forall (x,y) \in X \times X, \qquad d(x,y) \quad = \quad d'(f(x), f(y))$$

Lorsqu'un tel plongement existe, on dit alors que (X,d) est isométriquement plongeable dans (X',d').

D'autre part, on dispose de la distance bien connue d_{ℓ_p} sur $\mathbb{R}^n$ définie par :

$$\forall x, y \in \mathbb{R}^n, \qquad d_{\ell_p}(x,y) \quad = \quad \left(\sum_{i=1}^{n} |x_i - y_i|^p \right)^{\frac{1}{p}}$$

Dans la suite, on note : $\ell_p^m = (\mathbb{R}^m, d_{\ell_p})$, et ℓ_p^∞ l'espace des suites ℓ_p-convergentes. Et on dit que :

[2.J] Définition ***(ℓ_p-plongeabilité)***

Soit (X,d) un espace semi-métrique.
On dit que (X,d) est plongeable dans ℓ_p si (X,d) est plongeable isométriquement dans ℓ_p^m pour un certain $m \geq 1$. Le plus petit tel m est la dimension ℓ_p de X, notée $m_{\ell_p}(X,d)$.
(X,d) est plongeable dans ℓ_p^∞ si (X,d) est plongeable isométriquement dans ℓ_p^∞.

Intéressons-nous maintenant aux cas particuliers dont on se servira par la suite.

2.2.1 Plongements ℓ_1

Le problème de savoir si une semi-métrique donnée est plongeable dans ℓ_1 semble privé de tout mystère par le :

[2.K] Théorème ***(ℓ_1-plongeabilité)***

(V_n, d) est ℓ_1-plongeable si et seulement si $d \in Cut_n$.

Mais il faut prendre garde à ne pas conclure hâtivement que cela résout définitivement le problème de la ℓ_1-plongeabilité. En effet, nous avons dit que décider l'appartenance au cut cone est NP-complet. Ce théorème est donc bien sûr d'un intérêt théorique essentiel, mais il ne donne pas de critère efficace pour décider la ℓ_1-plongeabilité, malgré les apparences. Passons à sa preuve.

Preuve de [2.K].

- Supposons que (V_n, d) soit plongeable dans ℓ_1 :

On dispose par hypothèse de vecteurs $v_1, \ldots, v_n$ de $\mathbb{R}^m$ tels que $d(i,j) = \|v_i - v_j\|_1$, pour un certain $m \geq 1$.

Supposons pour le moment que $m = 1$. Alors $v_i \in \mathbb{R}$. Quitte à permuter les v_i, on peut supposer que $v_1 \leq v_2 \leq \ldots \leq v_n$. Alors, par téléscopage, on a :

$$d = \sum_{k=1}^{n-1} (v_{k+1} - v_k) \delta(\{1, 2, \ldots, k\})$$

Et donc $d \in \mathrm{Cut}_n$.

Maintenant, si $m > 1$, on fait la même manipulation sur chaque coordonnée. Les différentes coordonnées étant "indépendantes" dans le calcul de la norme ℓ_1, on peut sommer les semi-métriques de coupe obtenues sur chaque coordonnée pour arriver à la semi-métrique finale d recherchée. On exprime donc d comme somme de semi-métriques de coupe.

On notera que les permutations engendrées par les réordonnancements dans l'ordre croissant des différentes coordonnées peuvent ne pas être les mêmes, mais ce n'est pas un problème : l'important est que l'on arrive à exprimer la participation de chaque coordonnée à la distance d en termes de semi-métriques de coupes.

Finalement, on en déduit que dans tous les cas, $d \in \mathrm{Cut}_n$.

- Supposons que $d \in \mathrm{Cut}_n$:

On peut écrire par hypothèse :

$$d = \sum_{k=1}^{m} \lambda_k \, \delta(S_k)$$

où $\lambda_k \in \mathbb{R}_+$ et $S_k \subset V_n$, pour certains m et n fixés.

On définit simplement les vecteurs $v_1, \ldots, v_n$ de $\mathbb{R}^m$ ainsi :

$$(v_i)_k \quad = \quad \begin{cases} \lambda_k & \text{si } i \in S_k \\ 0 & \text{sinon} \end{cases}$$

Il est alors clair qu'on a $\|v_i - v_j\| = d_{i,j}$. Donc (V_n, d) est ℓ_1-plongeable. □

Un autre résultat de plongement dans ℓ_1 fondamental est le suivant :

[2.L] Théorème ***(plongeabilité de ℓ_2 dans ℓ_1)***

*L'espace ℓ_2^n se plonge avec distorsion constante (*i.e. *ne dépendant pas de n) dans ℓ_1^m, avec m polynomial en n.*

La preuve de ce théorème, bien que conceptuellement intéressante, sort du cadre de ce mémoire et on renverra le lecteur intéressé vers [Ind07] ou [Ber91].

2.2.2 Plongements ℓ_2

Ici, on se demande si une distance d sur V_n est euclidienne. Il faut donc trouver un moyen de demander à la distance d "d'imiter" le comportement des normes euclidiennes dont on a l'habitude. Posons tout d'abord la :

[2.M] Définition ***(Fonction de type négatif)***

Une fonction f sur V_n est dite de type négatif si :

$$\forall\, b \in \mathbb{Z}^n \quad \textit{tel que} \quad \sum_{i=1}^{n} b_i = 0, \qquad \sum_{1 \le i < j \le n} b_i\, b_j\, f(i,j) \le 0$$

On dispose alors du théorème :

[2.N] Théorème ***(ℓ_2-plongeabilité)***

(V_n, d) est plongeable dans ℓ_2 si et seulement si d^2 est une fonction de type négatif sur V_n.

On peut voir que la vérification du fait qu'une fonction est de type négatif peut se faire en temps polynomial. Ce résultat contraste donc avec le caractère NP-complet de décision du problème de ℓ_1-plongeabilité.

Bien sûr, ce qui va jouer un rôle essentiel dans la démonstration de ce théorème est qu'on a un produit scalaire canonique dans un espace ℓ_2.

Preuve de [2.N].

• Supposons que (V_n, d) soit plongeable dans ℓ_2. On dispose donc d'une famille $(v_i)_{1 \le i \le n}$ de vecteurs de $\mathbb{R}^m$, $m \ge 1$, tels que :

$$\forall i, j \in \{1, \ldots, n\}, \quad \|v_i - v_j\|_2 = d(i,j)$$

Calculons à présent, en prenant $b \in \mathbb{Z}^n$ tel que $\sum_{i=1}^n b_i = 0$:

$$\begin{aligned}
\sum_{1\leq i<j\leq n} b_i\, b_j\, d(i,j)^2 &= \sum_{1\leq i<j\leq n} b_i\, b_j\, \|v_i - v_j\|_2^2 \\
&= \frac{1}{2}\sum_{1\leq i,j\leq n} b_i\, b_j\, \|v_i - v_j\|_2^2 \\
&= \frac{1}{2}\sum_{1\leq i,j\leq n} b_i\, b_j\, (\|v_i\|_2^2 + \|v_j\|_2^2 - 2\,\langle v_i, v_j\rangle) \\
&= \sum_{j=1}^{n}\left(b_j \sum_{i=1}^{n} b_i\, \|v_i\|_2^2\right) - \sum_{1\leq i,j\leq n} \langle b_i\, v_i, b_j\, v_j\rangle \\
&= 0 - \|\sum_{i=1}^{n} b_i\, v_i\|_2^2 \\
&\leq 0
\end{aligned}$$

Et donc d^2 est de type négatif sur V_n.

• Supposons maintenant que d^2 soit de type négatif sur V_n. Il s'agit de construire le plongement dans un ℓ_2^m.

Commençons par considérer simplement l'application φ qui envoie $i \in V_n$ sur le vecteur e_i de la base canonique de $\mathbb{R}^n$. On cherche maintenant à former un produit scalaire sur $\mathbb{R}^n$ qui fera de φ un plongement isométrique.

On sait qu'on a, pour tout produit scalaire $\langle .,. \rangle$: $-2\,\langle e_i, e_j\rangle = \|e_i - e_j\|_2^2 - \|e_i\|_2^2 - \|e_j\|_2^2$.

Il est donc naturel de vouloir poser notre produit scalaire $\langle .,. \rangle_\star$ tel que $\langle e_i, e_j\rangle_\star = -\frac{1}{2}(d(i,j)^2 - 2)$. Posons donc la matrice A définissant notre candidat à être $\langle .,. \rangle_\star$:

$$a_{i,j} = -\frac{1}{2}(d_{i,j}^2 - 2)$$

Pour que A définisse un produit scalaire, il faut et il suffit qu'elle soit définie positive. Vérifions cette propriété. Soit $b \in \mathbb{R}^n$, alors :

$$\begin{aligned}
\sum_{1\leq i,j\leq n} b_i b_j a_{i,j} &= \sum_{1\leq i,j\leq n} b_i b_j (1 - \frac{1}{2} d_{i,j}^2) \\
&= \sum_{1\leq i,j\leq n} b_i b_j - \frac{1}{2}\sum_{1\leq i,j\leq n} b_i b_j d_{i,j}^2
\end{aligned}$$

Si on suppose que $b_i \in \mathbb{Z}$ et $\sum_{i=1}^n b_i = 0$, on a alors bien, par la propriété de type négatif de d^2 :

$$\sum_{1\leq i,j\leq n} b_i b_j a_{i,j} \geq 0$$

et donc A est positive. La restriction à $\mathbb{Z}$ n'est pas un problème, car par division on peut étendre cette propriété pour $b_i \in \mathbb{Q}$, et par densité pour $b_i \in \mathbb{R}$.

Donc A est positive sur l'hyperplan $H = \{b \in \mathbb{R}^n \mid \sum_{i=1}^n b_i = 1\}$. Elle est donc définie positive sur $E = H/\operatorname{Ker} A \simeq \mathbb{R}^{(\operatorname{rg} A)-1}$. Ainsi, en posant ψ l'application φ projetée (orthogonalement sur H, puis avec la surjection canonique sur E), on obtient le plongement attendu. □

2.3 Distorsion

2.3.1 Définition

La définition de plongement isométrique que nous avons déjà donnée est très stricte : les deux espaces mis en jeu sont exactement identiques au point de vue métrique. Mais il se trouve qu'un point de vue fertile pour la suite est de considérer les espaces qui *se ressemblent* au point de vue métrique, sans exiger qu'ils soient strictement *identiques*. Cela mène à introduire la notion de *distorsion*, en posant la :

[2.O] Définition ***(Plongement lipschitzien)***

Un plongement lipschitzien *de* (X,d) *dans* (X',d') *est une fonction* $f : X \longrightarrow X'$ *tel que, avec* $C \geq 1$ *la distorsion :*

$$\begin{array}{lccccc} \forall x,y \in X, & \frac{1}{C} d(x,y) & \leq & d'(f(x),f(y)) & \leq & d(x,y) \\ \text{ie.} & d'(f(x),f(y)) & \leq & d(x,y) & \leq & C\, d'(f(x),f(y)) \end{array}$$

2.3.2 Lemme topologique

(V_n,d) n'est pas toujours ℓ_1 plongeable. Cependant, il y a toujours un plongement de (V_n,d) dans un ℓ_1, où les distances sont conservées à un facteur multiplicatif près.

Nous allons tout de suite illustrer le fait que l'isométrie stricte est trop restrictive, mais qu'en s'autorisant une petite déformation, on peut identifier certains espaces métriques.

[2.P] Théorème ***(Plongement*** ℓ_1***, lemme de Bourgain)***

On peut plonger un espace métrique fini (X,d)*,* $|X| = n$*, dans* $(\mathbb{R}^N, \ell_1)$*, avec* $N = \sum_{p=1}^{\lfloor \log_2 n \rfloor} \binom{n}{2^p} < 2^n$*, avec une distorsion* $O(\log n)$*.*

Preuve de [2.P].

La démarche étant sensiblement la même dans toutes ses petites variations, on ne présente que la preuve du lemme original de Bourgain. L'idée du lemme de Bourgain est, comme pour la plupart des plongements d'espaces finis dans des espaces l_p ou l_∞, de remplacer chaque point par la fonction "distance à ce point" (l'isométrie pour la norme infinie résulte simplement de l'inégalité triangulaire). Mais comme il s'agit de rester dans des espaces de dimension finie, il faut discrétiser, et on se limite donc à la distance à certains ensembles bien choisis.

Construisons d'abord l'espace dans lequel on va plonger l'espace fini. Pour tout s dans $\{1,\ldots,n\}$, soit P_s la famille des sous-ensembles de X de cardinal s. On pose ensuite $P = \bigcup_{p=1}^{\lfloor \log_2 n \rfloor} P_{2^p}$. On a alors $|P| = N$, et nous allons considérer des vecteurs de $\mathbb{R}^{|P|} = \mathbb{R}^N$.

On construit donc les vecteurs $u_x \in \mathbb{R}^{|P|}$ en les indexant par les $A \in P$:

$$\forall A \in P, \quad u_x(A) = \frac{1}{\lfloor \log_2 n \rfloor \binom{n}{|A|}} d(x,A)$$

Vérifions d'abord que $\|u_x - u_y\|_1 \geq d(x,y)$ pour tous x et y dans X :

$$\begin{aligned} ||u_x - u_y||_1 &= \sum_{A \in P} |u_x(A) - u_y(A)| \\ &= \frac{1}{\lfloor \log_2 n \rfloor} \sum_{A \in P} \frac{|d(x,A) - d(y,A)|}{\binom{n}{|A|}} \\ &\leq \frac{1}{\lfloor \log_2 n \rfloor} \sum_{A \in P} \frac{d(x,y)}{\binom{n}{|A|}} \\ &= d(x,y) \end{aligned}$$

où l'on a utilisé que la fonction distance est 1-lipschitzienne.

Montrons maintenant que, pour x, y fixés, et pour une certaine constante c_0, on a :

$$c_0 \lfloor \log_2 n \rfloor \, \|u_x - u_y\|_1 \leq d(x,y)$$

Pour $z \in X$, on note $B(z,\rho)$ la boule ouverte de centre z de rayon ρ.

On définit ρ_t de la façon suivante : $\rho_0 = 0$ et $\forall t \geq 1$, t entier, ρ_t est le plus petit scalaire tel que $|B(x, \rho_t)| \geq 2^t$ et $|B(y, \rho_t)| \geq 2^t$, et $\rho_t < \frac{1}{2}d(x, y)$.

Soit $t^\star$ le plus grand de ces t, on pose $\rho_{t^\star+1} = \frac{1}{2}d(x, y)$. Ainsi, pour $t \in \{1, \ldots, t^\star + 1\}$, $B(x, \rho_t)$ et $B(y, \rho_t)$ sont disjointes.

Soient $t \in \{1, \ldots, t^\star + 1\}$ et ρ tel que $\rho_{t-1} \leq \rho \leq \rho_t$. On a :

$$|d(x, A) - d(y, A)| \geq \rho - \rho_{t-1}$$

pour tout sous-ensemble A de X vérifiant $A \cap B(x, \rho) = \varnothing$ et $A \cap B(y, \rho_{t-1}) \neq \varnothing$. D'où, en sommant :

$$\begin{aligned} \forall s \in \{1, \ldots, n\}, \frac{1}{|P_s|} \sum_{A \in P_s} |d(x, A) - d(y, A)| &\geq (\rho - \rho_{t-1}) \frac{|A \in P_s | A \cap B(x, \rho) = \varnothing, A \cap B(y, \rho_{t-1}) \neq \varnothing|}{|P_s|} \\ &= (\rho - \rho_{t-1})\mu_t \end{aligned}$$

où l'on a posé $\mu_t = \frac{|\{A \in P_s | A \cap B(x,\rho) = \varnothing, A \cap B(y,\rho_{t-1}) \neq \varnothing\}|}{|P_s|}$.

Il s'agit alors de remarquer que si l'on choisit s d'une façon bien précise, les μ_t sont minorés par une constante absolue μ_0 ne dépendant pas de t, x ou y, résultat un peu technique que l'on admettra avec $s = s_t := 2^p$ où p est le plus petit entier tel que $\frac{n}{10*2^t} \leq 2^p$. En réalité, on peut même dire $\mu_0 = e^{-\frac{8}{10}}(1 - e^{-\frac{1}{20}}) \approx 0.02$. On en déduit, comme les s_t correspondant à $t = 1 \ldots t^\star + 1$ sont distincts :

$$\frac{1}{|P_{s_t}|} \sum_{A \in P_{s_t}} |d(x, A) - d(y, A)| \geq (\rho - \rho_{t-1})\mu_0$$

On fait tendre ρ vers ρ_t :

$$\frac{1}{|P_{s_t}|} \sum_{A \in P_{s_t}} |d(x, A) - d(y, A)| \geq (\rho_t - \rho_{t-1})\mu_0$$

et il ne reste plus qu'à sommer le tout pour $t = 1 \ldots t^\star + 1$:

$$\begin{aligned} \lfloor \log_2 n \rfloor \|u_x - u_y\|_1 &\geq \sum_{t=1}^{t^\star+1} (\rho_t - \rho_{t-1})\mu_0 \\ &= \rho_{t^\star+1}\mu_0 \\ &= \frac{\mu_0}{2} d(x, y) \end{aligned}$$

Ce qui donne le résultat désiré avec $c_0 = \frac{2}{\mu_0}$.

Il est à noter qu'on peut facilement prouver un résultat similaire pour tous les espaces ℓ_p pour $p \geq 1$, en introduisant à la place des vecteurs u_x les vecteurs v_x définis par :

$$v_x(A) = \frac{1}{(\lfloor \log_2 n \rfloor \binom{n}{|A|})^{\frac{1}{p}}} d(x, A)$$

□

La borne logarithmique obtenue avec ce lemme topologique est optimale, puisqu'elle est atteinte pour une certaine classe de graphes, appelés graphes expanseurs. Nous en donnons simplement la définition ici par souci de complétude, mais nous ne les étudierons pas.

[2.Q] Définition ***(Graphes expanseurs)***

Soit $G = (V, E)$ un graphe fini. On définit l'expansion d'arêtes (edge expansion) de G par

$$h(G) = \min_{1 \leq |S| \leq \frac{|V|}{2}} \frac{|\partial(S)|}{|S|}$$

où $\partial(S)$ désigne le nombre d'arêtes avec une extrémité dans la coupe S.

Un graphe d-régulier ϵ-expanseur est alors un graphe tel que chaque sommet possède d arêtes et tel que l'expansion d'arêtes soit supérieure à $d \times \epsilon$.

Intuitivement, un graphe expanseur est un graphe dont chaque sous-ensemble (ou coupe) est "très" relié à son complémentaire, le "très" étant quantifié par la proportion des arêtes partant de S et allant dans le complémentaire de S.

Le théorème est alors le suivant [LLR95] :

[2.R] Théorème ***(Plongements de graphes expanseurs)***

Soit $G = (V, E)$ un graphe d-régulier ϵ-expanseur à n sommets ($k \geq 3, \epsilon \geq 0$).
Alors tout plongement de G dans un espace ℓ_p pour $1 \leq p \leq 2$ a une distorsion de l'ordre de $\log n$.
De plus, on ne peut le plonger avec distorsion constante dans un espace normé quelconque de dimension $o(\log^2 n)$.

Notons que la deuxième assertion se déduit de la première par le théorème de Dvoretzky ([Dvo61]) stipulant que tout espace normé de dimension d peut être plongé avec distorsion $\sqrt{d}$ dans un espace ℓ_2.

Malgré tout, il y a de nombreuses améliorations techniques du lemme de Bourgain [2.P] lorsqu'on change quelque peu les hypothèses. Ce qui suit en est un exemple et nous servira par la suite (la démonstration étant très similaire, on ne la présente pas ici) :

[2.S] Théorème ***(Raffinement utile)***

(X, d) métrique fini, $|X| = n$, $Y \subseteq X$, $|Y| = k$. Alors on a un plongement f dans $\mathbb{R}^K$, $K = \sum_{p=1}^{\lfloor lgk \rfloor} \binom{k}{2^p} < 2^k$, tel que :

$$\begin{array}{lrcl} \forall x, y \in X & ||f(x) - f(y)||_1 & \leq & d(x, y) \\ \forall x, y \in Y & d(x, y) & \leq & c_0 \log k \, ||f(x) - f(y)||_1 \end{array}$$

C'est-à-dire qu'il est possible de contracter un peu plus sur le complémentaire de Y.
On peut demander $K = O(n^2)$, voire $K = O(\log^2 n)$ (mais ce dernier est randomisé), et le plongement peut être trouvé en temps polynomial.

Il y a deux améliorations dans ce raffinement du lemme de Bourgain : d'une part la restriction au sous-ensemble Y, et d'autre part la réduction de la dimension de l'espace à des tailles polynomiales, voire mieux.

Pour se limiter à un sous-ensemble Y, on reprend le raisonnement du lemme de Bourgain, sauf que l'on prend pour les ensembles A des sous-ensembles de Y et non de X, les mêmes propriétés en découlent alors.

La dimension est réduite de façon astucieuse : l'algorithme pour trouver le plongement est quasiment le même que celui de Bourgain que nous avons présenté, sauf qu'au lieu de considérer tous les espaces de cardinal 2^p, on considère, pour chaque entier $t = 1, \ldots, \lfloor \log_2 n \rfloor - 1$, $O(\log n)$ ensembles choisis aléatoirement et de façon indépendante de cardinal $\frac{n}{2^t}$. On définit alors le plongement, pour chaque point x, comme le vecteur de taille $O(\log^2 n)$ dont chaque coordonnée vaut la distance de x à un des $O(\log^2 n)$ ensembles. Ensuite, tout marche comme précédemment (voir [LLR95] ou [AR98] pour une preuve précise).

3 Graphes et flots contraints

3.1 Préliminaire : Programmation linéaire

3.1.1 Objectif & enjeux

Un problème de programmation linéaire est la donnée d'une forme linéaire de $\mathbb{R}^d$ qu'on veut maximiser sur un certain polytope. C'est-à-dire que le domaine sur lequel on maximise la forme linéaire est délimité par des inégalités et égalités (linéaires en les coordonnées). On peut donc aussi voir la programmation linéaire comme la maximisation d'une forme linéaire sous certaines contraintes linéaires.

La programmation linéaire intervient dans beaucoup de problèmes, où elle constitue la brique de base du raisonnement. C'est en l'occurence le cas dans notre étude où nous l'utiliserons comme outil intermédiaire. Nous allons donc nous intéresser rapidement à quelques résultats la concernant, sans entrer dans les détails techniques.

3.1.2 Forme(s) d'expression du problème

Il y a plusieurs formes équivalentes d'expression d'un même problème de programmation linéaire. Cela permet une plus grande flexibilité dans la manipulation des instances du problème. Nous exposons ici la forme dite *standard*, qui nous suffira.

[3.A] Définition ***(Expression d'un problème de programmation linéaire en forme standard)***
Soient $d, m \geq 1$. Soient $c \in \mathbb{R}^d$, $b \in \mathbb{R}^m$ et A une matrice $m \times d$. Alors le problème de programmation linéaire associé sous forme standard est de trouver un vecteur $x \in \mathbb{R}^d$ qui vérifie :

$$\begin{array}{rlcl} \text{Maximiser} & \langle x, c\rangle & & \\ \text{Sous les contraintes} & Ax & \leq & b \\ & x & \geq & 0 \end{array}$$

De manière plus explicite, on peut reformuler ceci en :

$$\begin{array}{rlcll} \text{Maximiser} & \sum_{j=1}^{d} c_j\, x_j & & & \\ \text{Sous les contraintes} & \sum_{j=1}^{d} a_{i,j}\, x_j & \leq & b_i & , \text{ pour } i = 1 \dots m \\ & x_j & \geq & 0 & , \text{ pour } j = 1 \dots d \end{array}$$

Bien entendu, il est des cas où le maximum n'existe pas, *i.e.* la forme linéaire n'est pas bornée sur le domaine considéré ou le domaine est vide. Dans ce cas, le problème est dit *non borné*, ou *infaisable*, respectivement.

3.1.3 Complexité

Ce n'est *a priori* pas évident, mais il existe des algorithmes résolvant le problème de programmation linéaire. Il y en a principalement trois : l'algorithme du simplexe, l'algorithme de l'ellipsoïde et l'algorithme du point intérieur. Le premier est de complexité exponentielle alors que les deux suivants s'exécutent en temps polynomial.

Cependant, en pratique, on utilise plutôt l'algorithme du simplexe, qui s'exécute paradoxalement plus rapidement que ses homologues polynômiaux. L'idée sous-jacente à cet algorithme est que le maximum de la forme linéaire va être atteint en un sommet du polytope définissant le domaine. On se déplace donc de sommet en sommet de manière optimale jusqu'à arriver au bon.

Nous n'expliquerons pas ici le fonctionnement des algorithmes car ceci nous éloignerait trop de notre objet d'étude. Pour plus de détails sur l'algorithme du simplexe en particulier, et sur le problème de programmation linéaire en général, nous renvoyons à l'excellent [CLRS01].

3.1.4 Dualité

Le grand résultat de la programmation linéaire est certainement l'équation de dualité.

On définit le dual d'un problème de programmation linéaire sous forme standard en remplaçant dans [3.A] :

- $x \in \mathbb{R}^d$ par $y \in \mathbb{R}^m$
- *Maximiser* par *Minimiser*,
- $\langle c, x\rangle$ par $\langle b, y\rangle$
- $Ax \leq b$ par ${}^{\mathrm{t}}Ay \geq c$
- $x \geq 0$ par $y \geq 0$

On peut alors montrer que si $x \in \mathbb{R}^d$ est une solution optimale d'un problème de programmation linéaire, et si $y \in \mathbb{R}^m$ est une solution optimale du problème dual, alors :

$$\langle c, x\rangle \leq \langle b, y\rangle$$

Or il se trouve que l'algorithme du simplexe permet de résoudre à la fois le problème de départ et son dual, donnant les mêmes valeurs finales pour les fonctions objectifs. C'est donc une solution optimale et on a (lorsque ces quantités sont bien définies) :

[3.B] Théorème ***(Dualité en programmation linéaire)***
Soit $c \in \mathbb{R}^d$, $x \in \mathbb{R}^d$, $b \in \mathbb{R}^m$, $y \in \mathbb{R}^m$ et A une matrice $m \times d$. Alors :

$$\max_{\substack{x\geq 0 \\ Ax\leq b}} \langle c, x\rangle \quad = \quad \min_{\substack{y\geq 0 \\ {}^{\mathrm{t}}Ay\geq c}} \langle b, y\rangle$$

3.2 Multicommodity-flow

3.2.1 Position du problème

On désigne toujours par $G = (V, E)$ un graphe fini non-orienté. Pour chaque arête e, on fixe une capacité $C_e \in \mathbb{R}$.

Pour se faire une image, on peut penser que le graphe schématise le déplacement d'un fluide dans des canalisations. Chaque arête est un tuyau qui a un débit maximal de C_e. L'objectif de la majorité des algorithmes sur de tels graphes est de maximiser, sous certaines conditions, le flot total de liquide qu'on fait passer dans le graphe, avec plus ou moins de points d'entrée et de sortie de fluide selon les cas.

De tels graphes, appelés *réseaux* apparaissent dans toute une variété de problèmes et c'est pourquoi s'intéresser à l'efficacité d'algorithmes les concernant est essentiel dans beaucoup de domaines.

Nous définissons ici un cadre particulier : on se donne k *paires de commodités* $(s_1, t_1) \dots (s_k, t_k) \in V^2$ et pour chaque paire, on demande qu'il transite au minimum un débit D_h entre s_h et t_h. Dans le cadre de l'analogie qui précède, cela revient à considérer différents liquides devant tous passer d'un point précis à un autre et ne pouvant pas se mélanger. La question est de savoir quel est le débit total maximal qu'on peut faire transiter dans le réseau en respectant les contraintes, et de trouver une coupe du graphe (sous-ensemble des sommets) qui est limitante, c'est-à-dire trouver une coupure telle que les capacités à son interface empêchent d'augmenter le débit global sur le graphe.

Intuitivement, il apparaît assez évident que le débit global sur le graphe est soumis à une contrainte due aux capacités limitées de chaque arête. Formalisons cela.

On pose $\mathcal{P}_h = \{\text{chemins (ensemble d'arêtes) de } s_h \text{ à } t_h\}$ et $\mathcal{P} = \bigcup_{1\leq h\leq k} \mathcal{P}_h$ l'ensemble des chemins utiles.

On appelle **Flot Multicommodités** une fonction $f : \mathcal{P} \longrightarrow \mathbb{R}_+$. C'est un flot **faisable** pour l'instance (G, C, D) si :

$$\begin{array}{rcl} \forall e \in E, \quad \sum_{\substack{P\in\mathcal{P} \\ e\in P}} f(P) & \leq & C_e \\ \sum_{P\in\mathcal{P}_h} f(P) & \geq & D_h \end{array} \tag{3.i}$$

Cette formulation a l'avantage d'être limpide mais l'inconvénient de faire apparaître les chemins du graphe G qui sont en nombre exponentiel par rapport à n, le nombre de sommets de G.

Pour des raisons de complexité, il est essentiel d'avoir une formulation avec des contraintes polynômiales, de sorte que les algorithmes de programmation linéaire s'appliquent en temps polynomial en n. La formulation suivante est une formulation équivalente présentant cet avantage.

Les i permettent d'indexer les commodités, et les x^i_{uv} représentent le flot de la i-ème commodité traversant l'arête uv. On note s_i et t_i les points de départ et d'arrivée de la commodité i :

$$\begin{array}{lll} \forall e \in E & \sum_{1\leq i\leq k} x^i_e \leq C_e & \text{(les capacités limitent les flots)} \\ \forall 1 \leq i \leq k & \sum_{u\in V} x^i_{u,t_i} - \sum_{u\in V} x^i_{t_i,u} \geq D_i & \text{(flots} \geq \text{demandes)} \\ \forall 1 \leq i \leq k, v \in V | s_i \neq v \neq t_i, & \sum_{u\in V} x^i_{u,v} = \sum_{w\in V} x^i_{vw} & \text{(conservation des flots)} \\ \forall uv \in E, 1 \leq i \leq k, & x^i_{uv} \geq 0 & \end{array} \tag{3.ii}$$

On s'intéresse aux demandes maximales que l'on peut exiger tout en conservant l'existence d'un flot faisable. On veut donc déterminer le plus grand λ tel qu'il existe un flot multicommodités faisable pour l'instance $(G, C, \lambda D)$. On note cette valeur $\lambda^\star$ on dit que c'est le *max-flow*.

$\lambda^\star$ se trouve en résolvant le système de programmation linéaire défini par les équations (3.i) sur $(G, C, \lambda D)$, en ajoutant les conditions de positivité, et l'objectif étant de maximiser λ. Comme on l'a vu, on peut donc trouver la valeur de $\lambda^\star$ en temps polynomial.

3.2.2 Lien entre le Multicommodity flow et la coupe optimale

On voit facilement qu'un flot traversant le graphe, globalement, vérifie pour toute coupe $S \subseteq V$: flot traversant $S \leq$ capacité de S. Ainsi, si on définit :

$$\begin{aligned} cap(S) &= \sum_{e\in\delta_G(S)} C_e \\ dem(S) &= \sum_{h\in H_S} D_h \end{aligned}$$

où $\delta_G(S)$ est l'ensemble des arêtes ayant une extrémité dans S et l'autre dans $G \setminus S$, et H_S comme l'ensemble des commodités (s_h, t_h) qui ont une extrémité dans S et l'autre dans $G \setminus S$, on a, pour tout λ tel que $(G, C, \lambda D)$ admette un flot faisable :

$$\forall S \subseteq V, \quad \lambda\, dem(S) \leq cap(S)$$

et en particulier

$$\forall S \subseteq V, \quad \lambda^\star dem(S) \leq cap(S)$$

d'où :

$$\lambda^\star \leq \min_{S\subseteq V} \frac{cap(S)}{dem(S)}$$

Il y a en fait égalité pour $k = 1$ (cela se voit facilement, voir [CLRS01, p. 657]), et $k = 2$ [Rys63]. Pour $k > 2$, l'inégalité est stricte [DL97, p. 134].

Max-cut (optimisation) étant **NP-difficile**, trouver cette coupe S est également **NP-difficile**[2].

Une des principales applications de la théorie des plongements bi-lipschitiziens à faible distorsion est l'algorithme d'approximation des flots à plusieurs commodités sur un graphe, mis au point par Linial, London et Rabinovitch en 1994 [LLR95] et amélioré maintes fois depuis.

L'idée est au final plutôt simple : le problème se ramène à minimiser une fonction sur l'ensemble des métriques du graphe en question. On a une bonne caractérisation permettant de conclure pour celles qui

[2]Les deux problèmes sont reliés de la façon suivante. On prend un graphe G à n sommets et on construit un graphe $G^\star$ à $2n$ sommets en dédoublant le premier, en reliant chaque sommet à sa copie et en associant à chaque arête correspondante une capacité M, sans changer les capacités des autres arêtes dans les deux copies. Pour un tel graphe (en choisissant M assez grand), on voit facilement que si on sait calculer une coupe optimale $(A, \overline{A})$, alors $A \cap V(G), \overline{A} \cap V(G)$ constitue la max-cut de G. On se reporte à [MS90] pour une preuve détaillée.

sont plongeables dans ℓ_1 (comme combinaison linéaire des métriques de coupures), et donc on se ramène à celles-ci avec de la distorsion (d'ordre logarithmique) si c'est nécessaire.

Le théorème principal est le suivant :

[3.C] Théorème ***(Linial, London et Rabinovich, 1994)***

On a :

$$\lambda^\star \leq \min_{S\subseteq V} \frac{cap(S)}{dem(S)} \leq (c_0 \log k)\lambda^\star$$

où c_0 est une constante strictement positive universelle.
De plus, le sous-ensemble S en question peut être trouvé en temps polynomial.

On remarque que ce théorème contient deux choses différentes : d'une part il permet de majorer de façon assez efficace l'écart entre le *mincut* et le *maxflow*, d'autre part il permet explicitement de trouver une bonne approximation de la coupe optimale, pour des demandes et des capacités quelconques.

Preuve de [3.C].

On rappelle que $\lambda^\star$ est la solution du problème de programmation linéaire suivant :

$$\begin{array}{rlll} \text{Maximiser} & \lambda & & \\ \text{Sous les contraintes} & \forall e \in E, & \sum_{P\in\mathcal{P}|e\in P} f(P) & \leq C_e \\ & \forall 1 \leq h \leq k, & \sum_{P\in\mathcal{P}_h} f(P) & \geq \lambda\, D_h \\ & \forall P \in \mathcal{P} & f_P \geq 0 & \end{array}$$

qui se ramène par la dualité de la programmation linéaire au problème :

$$\begin{array}{rlll} \text{Minimiser} & \sum_{e\in E} C_e\, z_e & & \\ \text{Sous les contraintes} & \forall 1 \leq h \leq k, \forall P \in \mathcal{P}_h, & \sum_{e\in P} z_e & \geq y_h \\ & & \sum_{1\leq h\leq k} D_h y_h & \geq 1 \\ & \forall 1 \leq h \leq k & y_h \geq 0 & \\ & \forall e & z_e \geq 0 & \end{array}$$

On voit dans les contraintes qu'on peut supposer $y_h = \min_{P\in\mathcal{P}_h} \sum_{e\in P} z_e$ ce qui revient à voir y_h comme la distance de plus court chemin entre les deux extrémités de h sur le graphe G pondéré par les z_e. En généralisant ceci à toutes les arêtes du graphe, on nomme d_z la distance du plus court chemin du graphe pondéré par les z_e. On a alors :

$$\begin{aligned} \lambda^\star &= \sum_{e\in E} C_e z_e \\ &\geq \frac{\sum_{(i,j)\in E} C_{i,j}\, d_z(i,j)}{\sum_{h=1}^k D_h\, d_z(s_h,t_h)} \end{aligned}$$

car la somme au dénominateur est plus grande que 1 car $y_h = d_z(s_h,t_h)$ par définition, et $\sum_{1\leq h\leq k} D_h y_h \geq 1$.

Réciproquement, pour chaque semi-métrique d sur le graphe, en posant $z_{(i,j)} = \frac{d_{i,j}}{\sum_{h=1}^k D_h d_z(s_h,t_h)}$ et $y_h = \frac{d(s_h,t_h)}{\sum_{h=1}^k D_h d_z(s_h,t_h)}$, alors y et z vérifient les contraintes du programme, donc

$$\frac{\sum_{i,j\in E} C_{i,j}\, d(i,j)}{\sum_{h=1}^k D_h\, d(s_h,t_h)} \geq \lambda^*$$

Ainsi, la grandeur λ^* peut être réinterprétée comme

$$\lambda^* = \min_{d\in MET_V} \frac{\sum_{i,j\in E} C_{i,j}\, d(i,j)}{\sum_{h=1}^k D_h\, d(s_h,t_h)}$$

où le minimum porte sur l'ensemble MET_V, qui est l'ensemble de toutes les semi-métriques sur V

Si d est une semi-métrique de coupe $\delta(S)$, alors :

$$\frac{\sum_{i,j\in E} C_{i,j} d(i,j)}{\sum_{h=1}^{k} D_h d(s_h,t_h)} = \frac{cap(S)}{dem(S)}$$

Sinon, si d est plongeable isométriquement dans ℓ_1, on sait que $d = \sum_{S\subseteq V} \lambda_S \delta(S)$, d'où

$$\begin{aligned}\frac{\sum_{i,j\in E} C_{i,j}\, d(i,j)}{\sum_{h=1}^{k} D_h\, d(s_h,t_h)} &= \frac{\sum_S \lambda_S\, cap(S)}{\sum_S \lambda_S\, dem(S)} \\ &\geq \min_{S\subseteq V} \frac{cap(S)}{dem(S)}\end{aligned}$$

Sinon, on plonge notre graphe et sa métrique dans un espace ℓ_1 grâce au raffinement du lemme de Bourgain présenté précédemment en [2.S][3]. En prenant $X = V$ et $Y = \{s_h, t_h | h = 1 \dots k\}$, on obtient des vecteurs $(w_i)_{i\in V}$ dans un espace ℓ_1, ce qui donne :

$$\begin{aligned}\frac{\sum_{i,j\in E} C_{i,j} d(i,j)}{\sum_{h=1}^{k} D_h d(s_h,t_h)} &\geq \frac{\sum_{i,j\in E} C_{i,j} \|w_i - w_j\|_1}{(c_0 \log k)\sum_{h=1}^{k} D_h \|w_{s_h} - w_{t_h}\|_1} \\ &\geq \frac{1}{c_0 \log k} \min_{S\subseteq V} \frac{cap(S)}{dem(S)}\end{aligned}$$

Ainsi, $\lambda^\star \geq \frac{1}{c_0 \log k} \min_{S\subseteq V} \frac{cap(S)}{dem(S)}$

Maintenant, regardons comment construire le sous-ensemble S. On suit en réalité exactement le raisonnement de la preuve. On commence par calculer le *max-flow* $\lambda^\star$ en résolvant le programme linéaire dual exprimé en termes de semi-métriques (ce qui se fait en temps polynomial comme on l'a déjà vu).

Soit d une solution optimale. On plonge le graphe muni de la semi-métrique d dans un espace ℓ_1 sur des vecteurs w_i, ce qui peut se faire également en temps polynomial dans un espace de dimension polynômiale d'après [2.P]. On prend ensuite la projection de d sur une des coordonnées r telle que la grandeur :

$$\frac{\sum_{i,j\in E} C_{i,j} |w_i(r) - w_j(r)|}{\sum_{h=1}^{k} D_h |w_{s_h}(r) - w_{t_h}(r)|}$$

soit minimale. Comme elle est plongée dans ℓ_1 de dimension 1, la distance est maintenant facilement décomposable en métrique de coupures : $d = \sum_{A\in\mathcal{A}} \lambda_A \delta(A)$ comme on l'a vu en [2.K] (on s'est ramené en dimension 1 par la projection selon r). L'ensemble A_0 vérifiant $\frac{cap(A_0)}{dem(A_0)} = \min_{A\in\mathcal{A}} \frac{cap(A)}{dem(A)}$ vérifie alors également $\frac{cap(A_0)}{dem(A_0)} \leq c_0 \log k$. En effet :

$$\begin{aligned}\lambda^\star &= \frac{\sum_{i,j\in E} C_{i,j} d(i,j)}{\sum_{h=1}^{k} D_h\, d(s_h,t_h)} \\ &\geq \frac{\sum_{i,j\in E} C_{i,j} ||w_i - w_j||_1}{(c_0\, \log k)\sum_{h=1}^{k} D_h\, ||w_{s_h} - w_{t_h}||_1} \\ &= \frac{\sum_{A\in\mathcal{A}} \lambda_A cap(A)}{(c_0\, \log k)\sum_{A\in\mathcal{A}} \lambda_A\, dem(A)} \\ &\geq \frac{1}{c_0\, \log k} \frac{cap(A_0)}{dem(A_0)}\end{aligned}$$

□

[3] En fait, pour atteindre une distorsion en $\log k$ et pas $\log n$, on utilise le [2.S] avec le sous-ensemble $Y = \{s_h, t_h, h = 1...k\}$, et le plongement n'est bilipschitzien que sur ce sous-ensemble, ce qui nous suffit.

3.3 Le problème Sparsest-Cut

Le problème classique de Sparsest-Cut est déjà apparu dans le paragraphe précédent sans être clairement explicité. On reprend donc ici les notations du 2.2. On a défini la coupe limitante, ou *min-cut*, du Multi-commodity Flow comme étant la coupe S de $G = (V, E)$ minimisant la quantité $\Phi(S) = \frac{cap(S)}{dem(S)}$.

Le problème de Sparsest Cut consiste à déterminer cette coupe optimale et à calculer la constante $\Phi^\star$ (appelée constante de Cheeger) qui lui est associée. Précisément, on a :

$$\Phi^\star = \min_{S \subseteq V} \frac{\sum_{e \in \delta_G(S)} C_e}{\sum_{h \in H_S} D_h}$$

Le problème uniforme consiste à considérer des demandes D_h valant 1 entre chaque sommet du graphe et des capacités C_e valant toutes 1, on a alors

$$\Phi^\star = \min_{S \subseteq V} \frac{|E(S, V-S)|}{|S||V-S|}$$

où $E(S, V-S)$ désigne l'ensemble des arêtes reliant S à son complémentaire[4].

Le problème de Sparsest Cut est NP-difficile, mais n'en est pas moins fondamental en informatique. Par exemple, pour tous les algorithmes de type diviser pour régner, on cherche souvent à trouver des coupes de graphes permettant de séparer le graphe en deux avec un coût minimal (*i.e.* en coupant le moins d'arêtes possibles).

À défaut de disposer d'un algorithme polynomial pour résoudre ce problème, on cherche donc à développer des algorithmes d'approximation. Le théorème précédent en est un dans le sens où il présente un algorithme polynomial pour calculer une coupe et sa constante associée, approximant Sparsest Cut en $O(\log n)$.

Les méthodes d'approximation de Sparsest Cut fonctionnent pour la plupart par *relaxation*. Le calcul de $\Phi^\star$ peut en effet se reformuler, en notant toujours C (respectivement D) la fonction qui à une arête associe sa capacité (respectivement qui à une paire de points associe la demande entre les deux), de la façon suivante :

$$\begin{aligned} \Phi^\star &= \min_{S \subseteq V} \frac{\sum_{e \in \delta_G(S)} C_e}{\sum_{h \in H_S} D_h} \\ &= \min_{\delta} \frac{\sum_{e \in E} \delta(e) C_e}{\sum_{h \in H} \delta(h) D_h} \end{aligned}$$

où le minimum porte sur toutes les semi-métriques de coupure δ dans le graphe. Or l'ensemble des semi-métriques plongeables dans ℓ_1 est le cône engendré par ces semi-métriques de coupure, *i.e.* Cut_n d'après [2.K]. Donc on a :

$$\Phi^\star = \min_{d \in \mathrm{Cut}_n} \left\{ \sum_{i,j \in E} C_{i,j}\, d(i,j) \quad | \quad \sum_{i,j \in V^2} d(i,j)\, D_{i,j} = 1 \right\} \tag{3.iii}$$

où le minimum porte sur les distances plongeables dans ℓ_1. Calculer ce minimum algorithmiquement est délicat à cause des difficultés que pose le CUT-Cone, notamment les problèmes de décidabilité. C'est pourquoi on va chercher à l'approximer en relaxant le problème, c'est-à-dire en essayant de calculer ce minimum pour une classe plus large de métriques.

On peut ainsi réinterpréter plus simplement le résultat précédent : en relaxant la condition "d plongeable dans ℓ_1" en "d semi-métrique sur le graphe", on obtient le programme linéaire dual du programme permettant de calculer le plus grand flot à multi-commodités dans le graphe. Ceci met ainsi en évidence

[4] dans certains articles, on trouve aussi parfois

$$\Phi^\star = \min_{S \subseteq V \& |S| \leq \frac{|V|}{2}} \frac{|E(S, V-S)|}{|S|}$$

ce qui revient au même (en termes de complexité) même si la valeur est différente.

que la grandeur duale des flots n'est pas la coupe comme on pourrait s'y attendre mais bien les métriques sur le graphe. On peut relaxer sur d'autres ensembles de métriques, permettant d'obtenir des meilleures approximations pour Sparsest Cut. C'est l'objet de la troisième section.

3.4 Notre implémentation de l'algorithme

3.4.1 Objectifs et motivation

Le cœur de toute cette étude est de pouvoir rendre des problèmes NP-complets accessibles à la puissance de calcul de nos ordinateurs actuels, en trouvant des solutions approchées en temps polynomial.

Nous nous sommes donc demandés si tous ces travaux, qui paraissent très théoriques à première vue, remplissent effectivement leur rôle : rendre Sparsest-Cut plus accessible. La meilleure façon d'y répondre était d'implémenter réellement la démarche que nous avons décrite dans un programme de notre confection. Nous l'avons fait et présentons donc les résultats que nous avons obtenus.

Notre objectif a donc été de mettre en œuvre l'algorithme de London, Linial et Rabinovitch pour approximer le multi-commodity flow en $O(\log n)$. Nous n'avons en effet pas implémenté le raffinement, car notre objectif était de voir si la solution était viable, pas d'optimiser au maximum.

Nous avons réalisé l'application en C++, en utilisant la librairie 3D OpenGL.

3.4.2 Description du fonctionnement

La première étape consiste à transmettre le graphe au programme, et nous avons mis au point une syntaxe pratique et concise le permettant.

Ensuite, l'algorithme commence par calculer le *maxflow*. Pour ceci, il engendre toutes les équations du problème de programmation linéaire correspondant (vu en 3.ii). Il les transmet à une librairie extérieure nommée `lp_solve` (`http ://lpsolve.sourceforge.net/`), qui se charge de résoudre le problème à l'aide de l'algorithme du simplexe. On obtient alors la valeur $\lambda^\star$ du *maxflow* ainsi qu'un flot optimal faisable correspondant.

On munit alors le graphe de la distance définie par le flot. Pour cela, on utilise l'algorithme classique de Ford-Fulkerson (qu'on peut retrouver comme de nombreux autres algorithmes sur les graphes dans [Sed02]). Cet algorithme permet en effet de calculer efficacement (en $O(n^3)$) une matrice contenant la distance entre n'importe quel couple de points. Il fonctionne sur un principe de relaxation par rapport aux arêtes qu'on ne détaillera pas ici.

Cela permet de calculer le plongement dans ℓ_1 dont il est question à la fin de la preuve de [3.C]. On utilise donc un algorithme randomisé, qui utilise le générateur de nombres pseudo-aléatoires canonique disponible sur tout ordinateur. La complexité est clairement en $O(n \log n)$.

Après projection comme décrit ci-dessus, on décompose la semi-métrique uni-dimensionnelle comme somme de métriques de coupure. Cela fait simplement intervenir un tri sur les coordonnées, donc en $O(n \log n)$.

Si on suit le cheminement de la fin de la preuve de [3.C], il reste alors à trouver un minimum sur l'ensemble des ensembles créés, qui sont en nombre $O(\log^2 n)$.

Finalement, on obtient la *mincut* tant convoitée. On remarque au passage que la complexité de l'algorithme est très largement dominée par la résolution du système de programmation linéaire.

3.4.3 Les résultats obtenus

Afin d'illustrer le caractère très visuel de cet algorithme, notre programme affiche une représentation claire et parlante des graphes manipulés, des flots optimaux correspondants, ainsi que de la *mincut*. Voici quelques captures d'écran qui témoignent de cet effort :

Comme on peut le voir, il y a une fenêtre de rendu, qui affiche le graphe à proprement parler, et une fenêtre de type "console" qui informe de l'avancement du calcul, et des résultats. On peut faire afficher plus de résultats que ce qui est montré ici dans cette dernière fenêtre (valeurs exactes du flot, etc.) mais pour plus de clarté cela n'apparaît pas ici.

De même, il y a des moyens de mieux visualiser le graphe : on peut se déplacer librement autour du graphe pour mieux en cerner les propriétés, on peut redistribuer aléatoirement l'emplacement des

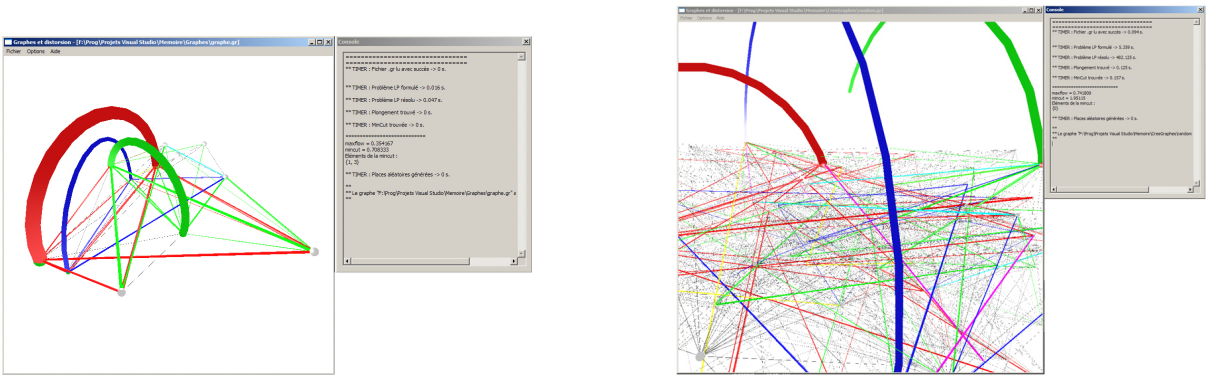


FIG. 3 – Notre programme illustrant l'algorithme de London, Linial et Rabinovitch

sommets pour voir le graphe sous un autre aspect, et on peut même projeter le graphe en 2 dimensions pour en avoir une vue plus synthétique.

Enfin, les couleurs indiquent la présence ou non d'un flot. On a fixé le nombre de commodités à trois. On a donc trois liquides distincts qui s'écoulent dans les tubes : un rouge, un vert et un bleu. Selon la proportion de chacun dans chaque tube, la couleur varie. La taille des tubes entre les sommets reflète également le débit qui y circule. Le flot représenté est le *maxflow* tel qu'il est calculé par le système de programmation linéaire. Les grands arcs qui s'élèvent au-dessus du graphe indiquent les trois paires de commodités.

Pour finir, il est possible de mettre en surbrillance les ensembles de la *mincut* que le programme calcule.

Au niveau des résultats expérimentaux, ils correspondent bien à nos prévisions : c'est la résolution du système de programmation linéaire qui est extrêmement coûteuse en temps. Le reste de l'algorithme s'exécute relativement rapidement. Notons que nous n'avons pas cherché à optimiser sauvagement le code, il est certainement possible de gagner un petit peu de temps en faisant vraiment attention (en particulier, nous n'avons pas cherché à optimiser le nombre d'équations du problème de programmation linéaire que l'on stipule, et il y a clairement des moyens de le réduire drastiquement).

Nous présentons ci-dessous un tableau récapitulant les résultats que nous avons obtenus sur plusieurs graphes tirés au hasard, en faisant varier le nombre de sommets et d'arêtes du graphe. À la lecture de ce tableau, il faut garder à l'idée qu'il y a quelques années, il était impensable de pouvoir trouver ainsi une approximation, même grossière, de la *mincut* d'un graphe à 100 sommets et 1000 arêtes.

Noeuds / Arêtes	**200**	**500**	**1000**
30	0.84 s	1.01 s	1.05 s
50	8.51 s	10.1 s	9.72 s
75	49.2 s	46.5 s	55.6 s
100	94 s	165.1 s	179.5 s

C'est pourquoi la recherche actuelle essaie de trouver des moyens d'éviter le passage par la programmation linéaire en essayant de trouver des solutions plus astucieuses.

4 Techniques d'approximation de Sparsest Cut

4.1 Préliminaire : Programmation semi-définie

La programmation semi-définie consiste en une généralisation de la programmation linéaire. Il s'agit comme en programmation linéaire de chercher un maximum d'une fonction linéaire. La différence est que ce maximum ne porte plus sur des conditions linéaires mais quadratiques. Géométriquement, cela signifie que l'on ne travaille plus sur un polytope mais sur un convexe de type quadrique.

Les problèmes de programmation semi-définie (SDP) s'expriment généralement en termes de matrices. On utilise les notations suivantes : pour A et B deux matrices, on introduit le produit scalaire de *Frobénius* $\langle A, B\rangle = Tr({}^{\mathrm{t}}AB)$; et on note $X \succeq 0$ pour X forme quadratique positive (semi-définie en anglais, d'où le nom de la technique). On note $A \succeq B$ pour $A - B \succeq 0$.

[4.A] Définition ***(Expression d'un problème de programmation semi-définie en forme standard)***

Soit C et $(A_i)_{1\leq i\leq m}$ dans $\mathbb{S}_n$ (ensemble des matrices symétriques $n \times n$) et des réels b_i. Résoudre le problème de programmation semi-définie signifie trouver un X qui vérifie :

$$\begin{array}{rl} \text{Maximiser} & \langle C, X\rangle \\ \text{Sous les contraintes} & \forall i \in [\![1, m]\!], \quad \langle A_i, X\rangle = b_i \\ & X \in \mathbb{S}_n \\ & X \succeq 0 \end{array}$$

La condition X positive est bien quadratique alors que la fonction à optimiser est bien linéaire, car le produit scalaire l'est. On peut également voir la programmation semi-définie comme une généralisation de la programmation linéaire avec une infinité de variables : la condition $X \succeq 0$ est équivalente à la condition $\forall x \in \mathbb{R}^n, {}^{\mathrm{t}}xXx \geq 0$. En pratique, la plupart des programmes cherchant à maximiser une fonction quadratique soumise à des contraintes quadratiques peuvent être réexprimés en des programmes semi-définis, et on ne passe pas toujours par cette formulation matricielle qui n'est là que pour formaliser les choses.

Tout comme en programmation linéaire, un résultat phare de la programmation semi-définie est l'équation de dualité, qui est, avec cette formulation matricielle du problème, la suivante :

[4.B] Théorème ***(Dualité en programmation semi-définie)***

Soit C, $A_i \in \mathbb{S}_n$ et $b \in \mathbb{R}^m$. Alors :

$$\max_{\substack{X\in\mathbb{S}_n,\ X\succeq 0 \\ \forall i\in[\![1,m]\!],\ \langle A_i,X\rangle=b_i}} \langle C, X\rangle = \min_{\sum_{i=1\dots m} y_i\, A_i \succeq C} \langle b, y\rangle$$

Si l'algorithme du simplexe ne trouve pas vraiment d'équivalent en programmation semi-définie, les algorithmes déjà vus pour la programmation linéaire, comme celui de l'ellipsoïde ou du point intérieur, se transposent très bien (et peuvent même se généraliser au cadre encore plus large de l'optimisation convexe). Cela implique en particulier que les programmes semi-définis se résolvent en temps polynomial.

4.2 Meilleures approximations de Sparsest Cut

4.2.1 Méthode générale par relaxation

Comme expliqué en 2.3, on peut trouver des meilleures approximations que celle en $O(\log n)$ de la deuxième partie en travaillant sur les relaxations de 3.iii. L'outil de la programmation semi-définie permet un choix plus large de catégories de métriques (ou même de fonctions qui n'en sont pas, i.e. ne vérifient pas l'inégalité triangulaire), puisque l'on sait également calculer des minima en temps polynomial lorsque les contraintes sont quadratiques.

En notant K le cône des fonctions de type négatif[5], une première approche fructueuse a été d'étudier les relaxations sur le cône K, mais la relaxation qui donne le plus de résultats est celle où l'on considère les métriques de type négatif (ensemble communément noté ${\ell_2}^2$ car ce sont les carrés de métriques plongeables dans ℓ_2), le programme correspondant est alors le suivant :

$$\begin{array}{rl} \text{Minimiser} & \sum_{u,v\in E} C_{u,v}||x_u - x_v||_2^2 \\ \text{Sous les contraintes} & \forall u \in V, x_u \in \mathbb{R}^n \\ & \sum_{u,v\in V} D(u,v)||x_u - x_v||_2^2 = 1 \\ & \forall (u,v,w) \in V^3, ||x_u - x_v||_2^2 \leq ||x_u - x_w||_2^2 + ||x_w - x_v||_2^2 \end{array}$$

Ce qu'il est important de considérer lors d'une relaxation est le saut d'intégralité (*integrality gap* en anglais), c'est l'écart entre le minimum relaxé et celui non relaxé (formellement c'est le quotient des deux).

C'est là qu'interviennent les plongements dans ℓ_1 : si la catégorie de métriques sur laquelle on relaxe se plonge isométriquement dans ℓ_1, le saut d'intégralité vaut alors 1, puisque $||x_u - x_v||_2^2 = ||\phi(x_u) - \phi(x_v)||_1$. Si ce n'est pas le cas, c'est alors la distorsion la plus faible avec laquelle on peut plonger cette catégorie de métriques dans ℓ_1 qui intervient, car elle borne naturellement le saut d'intégralité. On montre même qu'elles sont en fait égales, ce qui fait un lien élégant entre des considérations purement mathématiques (les plongements d'ensembles métriques finis dans des espaces ℓ_p) et un domaine purement algorithmique. On note ϕ un plongement de l'ensemble des x_u dans ℓ_1 avec distorsion c, on a donc

$$||x_u - x_v||_2^2 \leq d(\phi(x_u), \phi(x_v)) \leq c \times ||x_u - x_v||_2^2$$

Donc sous les contraintes ci-dessus, la grandeur $\sum_{u,v\in E} C_{u,v}||x_u - x_v||_2^2$ est toujours supérieure à $\frac{1}{c}$ fois la valeur du minimum sur les distances plongeables dans ℓ_1. Si elle lui est égale, alors comme somme de termes positifs, chacun des $||x_u - x_v||_2^2$ est égal à un $d(u,v)$ pour d plongeable dans ℓ_1, et donc la distance est en fait plongeable dans ℓ_1.

Ainsi, il suffit de développer un bon algorithme (*i.e.* avec faible distorsion) qui plonge les espaces finis munis d'une métrique de type négatif dans les espaces ℓ_1 (ou ℓ_2 puisque l'on peut plonger ℓ_2 dans ℓ_1 avec distorsion constante) pour avoir un algorithme d'approximation de Sparsest Cut de même complexité que la distorsion considérée.

Malheureusement, on n'a pas encore trouvé de borne stricte pour ces plongements. La conjecture de Goemans-Linial voudrait qu'il existe un tel plongement en $O(1)$, mais on y a trouvé des contre-exemples explicites, par exemple dans [LN06] et [KV05]. Il est conjecturé maintenant que le cube de Hamming, c'est-à-dire le cube à 2^d points de ℓ_1 est l'ensemble de points "le moins euclidien" possible d'une métrique de type négatif, et donc sa distorsion optimale est la pire que l'on puisse avoir. Cette conjecture donnerait ainsi une borne en $O(\sqrt{\log n})$ pour cette technique d'approximation de Sparsest Cut.

En fait, les recherches actuelles ont permis de trouver un plongement très proche, en $O(\sqrt{\log n}\log\log n)$, et d'atteindre cette borne optimale pour le problème Sparsest Cut uniforme (toutes les capacités valent 1 et les demandes sont uniformes et valent 1).

4.2.2 Meilleures approximations à ce jour

On s'intéresse d'abord dans ce paragraphe à l'approximation du problème de Sparsest Cut uniforme, qui est de calculer $\Phi^* = \min_{S\subseteq V} \frac{|E(S,V-S)|}{|S||V-S|}$.

Comme on vient de le voir, en relaxant le programme 3.iii à toutes les métriques de type négatif, le saut d'intégralité obtenu est exactement le même que la meilleure distorsion avec laquelle on peut plonger ces métriques dans un espace ℓ_1. L'algorithme en $O(\sqrt{\log n})$ ne fonctionne pas exactement de cette façon-là (et on ne sait toujours pas s'il existe un plongement de ${\ell_2}^2$ dans ℓ_1 de distorsion $\sqrt{\log n}$) mais trouve une bonne approximation de Sparsest Cut dans le cas uniforme en exploitant un résultat particulier sur la structure de l'espace ${\ell_2}^2$ des métriques de type négatif. Commençons par définir les concepts importants :

[5] C'est bien une relaxation car les distance ℓ_1 sont de type négatif, puisque les distances de coupure le sont (elles sont égales à leur carré qui sont des métriques d'espaces à deux points, donc plongeables isométriquement dans ℓ_2)

[4.C] Définition ***(Définitions nécessaires)***

Soit $G = (V, E)$ un graphe fini. On appelle une ${\ell_2}^2$-représentation de G un plongement de G dans un espace ${\ell_2}^2$, c'est-à-dire qu'on munit G d'une métrique de type négatif. C'est une ℓ_2^2-représentation unité si chaque vecteur est de norme 1. Une telle représentation est dite c-étendue si

$$\sum_{i<j} |v_i - v_j|^2 \geq 4c(1-c)n^2$$

Soient $v_1 \dots v_n \in \mathbb{R}^d$ et $\Delta \geq 0$, deux ensembles de vecteurs disjoints sont Δ-séparés $\forall v_i \in S, \forall v_j \in T, |v_i - v_j|^2 > \Delta$.

Le théorème de structure sur les espaces ${\ell_2}^2$ est alors le suivant([ARV04]) :

[4.D] Théorème ***(Coupes Δ-séparées d'espaces ${\ell_2}^2$)***

Pour tout $c > 0$, toute ${\ell_2}^2$ représentation unité c-étendue à n points contient des sous-ensembles Δ-séparés S et T de cardinaux $\Omega(n)$, où $\Delta = \Omega(\frac{1}{\sqrt{\log n}})$. De plus, il y a un algorithme randomisé polynomial en temps pour trouver ces sous-ensembles S et T.

La valeur de Δ de ce théorème est optimale comme le montre l'exemple simple de l'hypercube $\{-1; 1\}^d$: son plongement naturel forme une représentation ${\ell_2}^2$, et on montre (avec l'inégalité isopérimétrique des hypercubes) que deux sous-ensembles quelconques de taille $\Omega(n)$ contiennent forcément deux vecteurs dont la distance au carré est en $O(\frac{1}{\sqrt{\log n}})$.

Ainsi, on peut voir ce théorème comme une façon de dire que toutes les métriques ${\ell_2}^2$ c-étalées pour c constante sont "comme des hypercubes", ce qui est un grand pas en avant dans la compréhension de ces métriques.

Nous ne rentrerons pas dans le détail de la preuve très technique de ce théorème, mais ses applications pour approximer Sparsest Cut sont claires : il s'agit pour résoudre ce problème de trouver une coupe S du graphe G telle que S et $\overline{S}$ soient le plus espacés possibles (au sens qu'il y a le moins d'arêtes possibles qui les rejoignent), et donc, après relaxation, il s'agit de trouver une coupe telle que chaque point de la coupe soit le plus éloigné possible d'un point de son complémentaire. C'est exactement ce que nous fournit ce théorème.

Ainsi, l'algorithme d'approximation fonctionne comme suit : on résout en temps polynomial l'équation de programmation semi-définie, ce qui nous fournit une représentation ${\ell_2}^2$ du graphe et une valeur β solution du problème relaxé. On trouve, à l'aide de l'algorithme, deux sous-ensembles S et T Δ-séparés de taille $\Omega(n)$, et ceux-ci fournissent en repassant dans le graphe une coupe S vérifiant $E(S, \overline{S}) = O(|S|\,|\overline{S}|\beta\sqrt{\log(n)})$, montrant que le saut d'intégralité est en fait un $O(\sqrt{\log n})$.

Si cette technique ne s'applique qu'au cas uniforme, elle fournit néanmoins un résultat de structure des métriques ${\ell_2}^2$ très intéressant, qui a été utilisé par Arora, Lee et Naor pour fournir le meilleur algorithme d'approximation de Sparsest Cut à ce jour, basé précisément sur la relaxation énoncée au paragraphe précédent. Voici le résultat (nous renvoyons à [ALN08] pour la preuve) :

[4.E] Théorème ***(Plongement dans ℓ_2 en $O(\sqrt{\log n}\log\log n)$)***

Pour tout espace fini (X, d) muni d'une métrique de type négatif, il existe un plongement de (X, d) dans ℓ_2 de distorsion $O(\sqrt{\log n}\log\log n)$.

On remarque que le plongement fourni n'est qu'un plongement dans ℓ_2, et non dans ℓ_1. Mais on sait que le premier se plonge avec distorsion constante dans le second, en gardant des dimensions polynomiales([2.K]), donc on a bien le :

[4.F] Corollaire ***(Approximation en $O(\sqrt{\log n}\log\log n)$)***

Il existe un algorithme polynomial d'approximation de Sparsest Cut en $O(\sqrt{\log n}\log\log n)$.

Remarquons également que la conjecture stipulant que le cube de Hamming est le sous-ensemble de ℓ_1 le moins euclidien des espaces munis d'une métrique de type négatif est ici presque démontrée : modulo le facteur en logarithme itéré, la distorsion obtenue est précisément celle de l'hypercube $\{-1;1\}^d$. Les auteurs de l'article déclarent d'ailleurs qu'avec une preuve beaucoup plus technique, on peut probablement se débarrasser de ce facteur encombrant.

4.3 Conclusion et horizons de recherche

Les techniques présentées dans ce mémoire sont somme toute assez naturelles : la notion de distorsion permet de quantifier le caractère plus ou moins euclidien (ou plus ou moins ℓ_1) de certains espaces finis. Or, c'est précisément cet aspect des espaces (ou des métriques) qui rentre en compte lorsque l'on relaxe un problème qui utilise ces métriques. Si les résultats de plongement sont pour la plupart assez techniques, ils viennent s'appliquer avec élégance dans la démarche d'approximation de Sparsest Cut et son lien avec les Multi-commodity flows. Il reste néanmoins un certain nombre d'interrogations sur toutes ces questions.

À part pour le lemme de Bourgain, aucune borne de distorsion rencontrée dans ce mémoire n'est optimale : pour les métriques de type négatif, la distorsion optimale pour être plongé dans ℓ_2 est conjecturée comme étant $O(\sqrt{\log n})$, et on arrive à démontrer ce résultat modulo un facteur en logarithme itéré. Pour les plonger dans ℓ_1 on a longtemps cru que l'on pouvait y arriver à distorsion constante, mais les recents travaux de [LN06] et [KV05] ont permis de montrer qu'il y avait au moins une borne inférieure de distorsion en $O(\log\log n)^\delta$ pour tout $\delta > 0$, et même, en supposant vraie une conjecture classique appelée Unique Games Conjecture, une borne inférieure en $O(\log\log n)$.

Tous les algorithmes utilisant de la programmation semi-définie ont beau être polynomiaux, ils sont difficiles à mettre en pratique car extrêmement lents. [AKK04] ont récemment développé à partir des travaux de Arora, Rao et Vazirani une technique pour approximer Sparsest Cut en $O(\sqrt{\log n})$ avec un algorithme tournant en $\widetilde{O}(n^2)$ (donc sans résoudre de programme semi-défini), mais seulement dans le cas uniforme. La question de savoir si c'est possible pour le cas général reste ouverte.

On ne s'est intéressé dans cet exposé qu'aux algorithmes pour approximer Sparsest Cut dans le cas de graphes quelconques. Il y a évidemment de meilleurs algorithmes lorsque les graphes ont des caractéristiques particulières, notamment parce que ces graphes particuliers se plongent souvent "mieux" dans les espaces ℓ_p. Citons ici à titre non exhaustif le cas des *arbres*, qui peuvent être plongés dans ℓ_2 avec distorsion $O(\sqrt{\log\log n})$, borne atteinte pour les arbres binaires complets, et celui des *graphes planaires* que l'on peut plonger dans ℓ_2 avec distorsion $O(\sqrt{\log n})$. Mais la question des plongements dans ℓ_1 est bien plus délicate et fait encore l'objet de conjectures. Un domaine de recherche actif dans le sujet est la Planar multi-flow conjecture, énonçant qu'il existe une constante universelle C telle que tout graphe planaire admette un plongement dans ℓ_1 avec distorsion C.

Références

[AKK04] S. Arora, E. Kazan, and S. Kale. $O(\sqrt{logn})$ approximation to SPARSEST-CUT in $\widetilde{O}(n^2)$ time. *45th Annual Syposium on Foundations of Computer Science*, 2004.

[ALN08] S. Arora, J. Lee, and A. Naor. Euclidean distorsion and the Sparsest Cut. *Proceedings of the thirty-seventh annual ACM symposium on Theory of computing*, 2008.

[AR98] Y. Aumann and Y. Rabani. An O(log k) approximate min-flow max-cut theorem and approximation algorithm. *SIAM J. Comput.*, 27(1) :pp. 291–301, 1998.

[ARV04] S. Arora, S. Rao, and U. Vazirani. Expander flows, geometric embeddings, and graph partitionings. *36th Annual Symposium on the Theory of Computing*, 2004.

[Ber91] B. Berger. The fourth moment method. *SODA*, 2 :373–383, 1991.

[Bou85] J. Bourgain. On lipschitz embedding of finite metric spaces in Hilbert space. *Israel Journal of Mathematics*, 52 :pp. 46–52, 1985.

[Car07] O. Carton. Cours Langages formels, Calculabilité et Complexité. ENS, 2007.

[CLRS01] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to algorithms – 2nd edition.* MIT-Press, 2001.

[DL97] M.M. Deza and M. Laurent. *Geometry of cuts and metrics.* Springer, 1997.

[Dvo61] A. Dvoretzky. Some results on convex bodies and Banach spaces. *Proc. Internat. Symp. Linear Spaces*, 1961.

[Goe97] M. Goemans. Semidefinite programming in combinatorial optimization. *Math. Programming*, 79 :pp. 143–161, 1997.

[Ind07] P. Indyk. Uncertainty Principles, Extractors, and Explicit Embeddings of L_2 into L_1. *39th ACM Symposium on Theory of Computing*, 2007.

[KV05] S. Khot and N. Vishnoi. The unique games conjecture, integrality gap for cut problems and embeddability of negative type metrics into ℓ_1. *Proceedings of the 46th Annual IEEE Conference on Foundations of Computer Science*, 2005.

[LLR95] E. London, N. Linial, and Y. Rabinovitch. The geometry of graphs and some of its algorithmic applications. *Combinatorica*, 15(2) :pp. 215–245, 1995.

[LN06] J. Lee and A. Naor. L_p metrics on the Heisenberg group and the Goemans-Linial conjecture. *47th Annual IEEE Symposium on Foundations of Computer Science*, 2006.

[MS90] D. W. Matula and F. Shahrokhi. Sparsest cuts and bottlenecks in graphs. *Discrete Applied Mathematics*, 27 :pp. 113–123, 1990.

[Rys63] H.J. Ryser. Combinatorial Mathematics. *The Carus Mathematical Monographs*, 14, 1963.

[Sed02] R. Sedgewick. *Algorithms in C++ – 3rd edition*, volume 2 (Part 5). Addison Wesley, Princeton, 2002.