ALGORITHMES D'APPROXIMATION *

Il ne savait même pas combien il était d'approximatif...

Victor Chepoi et Yann Vaxes

Quelques notions préliminaires.

Soit NP la classe des problèmes de decision pouvant être validés par un algorithme non-déterministe polynomial et soit P la classe des problèmes de décision pouvant être validés par un algorithme déterministe polynomial ; NPC est la classe des problèmes NP-complets.

Un grand nombre de problèmes de décision ont une variante d'optimisation. Un *problème d'optimisation* Π est défini par :

–un ensemble d'instances D_Π;
–pour chaque instance $I \in D_\Pi$ il y a un ensemble des *solutions admissibles* $S_\Pi(I)$;
–une *fonction objectif* f_Π calculable en temps polynomial pour chaque couple (I, s) avec $I \in D_\Pi$ et $s \in S_\Pi(I)$;
–Π est soit un problème de *minimisation* soit un problème de *maximisation*.

Une *solution optimale* d'une instance I d'un problème d'optimisation Π est la solution admissible que minimise (maximise) la fonction objectif f_Π. Soit $\mathrm{OPT}(i)$ la valeur de fonction f_Π d'une solution optimale.

Un algorithme d'approximation est un algorithme polynomial qui produit des solutions proches de l'optimum. Soit Π un problème de minimisation (maximisation) et soit δ un nombre réel positif $\delta \geq 1$ ($\delta \leq 1$). Un algorithme $\mathcal{A}$ est un *algorithme d'approximation avec un facteur* δ pour Π si pour chaque instance $I \in \Pi$ l'algorithme $\mathcal{A}$ produit une solution admissible $s \in S_\Pi(I)$ telle que $f_\Pi(I, s) \leq \delta \cdot \mathrm{OPT}(I)$ ($f_\Pi(I, s) \geq \delta \cdot \mathrm{OPT}(I)$ si Π est un problème de maximisation).

Cours I. Quelques tubes des années 70

1.1. Ordonnancement de tâches (Minimum Makespan Scheduling)

> Le premier objet du Conseil est de répartir le travail indispenseble au-dedans et au-dehors de la maison.... Les garçons tondraient le gazon et ratisseraient les feuilles. Les filles balaieraient, épousseteraient et feraient la vaisselle du dîner. Tout le monde, excepté Papa, ferait son lit et rangerait sa chambre...
>
> *Treize à la douzaine* E. et F. Gilbreht

Formulation : Soient données m machines identiques et n tâches de durées $p_1, p_2, \ldots, p_n$, trouver une affectation des n tâches aux m machines telle que le temps de terminaison (*makespan*) soit minimum.

Algorithme LSA (*List Scheduling Algorithm*)

1. `Ordonner les tâches de façon arbitraire.`
2. `Exécuter la tâche courrante sur la machine la moins utilisée jusqu'ici.`

Théorème 1.1. *L'Algorithme LSA est un algorithme d'approximation avec un facteur 2 pour le problème d'ordonnancement de tâches.*

Preuve. Soit OPT le temps optimal et soit TEMPS le temps d'exécution des tâches par l'Algorithme LSA. On peut donner deux bornes inférieures pour OPT :

1. le temps moyen $(\sum_{i=1}^{n} p_i)/m$ de fonctionnement d'une machine ;

*Dans cette version très preliminaire il manque : tous les dessins, certains résultats importants, certains références et certains accents

2. la durée de la tâche la plus longue.

Pour la suite, soit $LB = max\{\frac{1}{m}\sum_i p_i, max\{p_i\}\}$. Soit M_i la machine dont le temps de fonctionnement est égal à TEMPS et soit j la dernière tâche exécutée sur M_i. On va noter par debut_j le début de la tâche j. Comme l'algorithme affecte la tâche à la machine la moins occupée, toutes les machines sont occupées à l'instant debut_j. Alors,

$$\text{debut}_j \leq \frac{1}{m}\sum_i p_i \leq \text{OPT}.$$

Puisque $p_j \leq OPT$, on obtient

$$\text{TEMPS} = \text{debut}_j + p_j \leq (\sum_{i=1}^{n-1} p_i)/m + p_j \leq 2 \cdot \text{ OPT}. \quad \square$$

Le pire exemple pour l'Algorithme LSA est constitué par m^2 tâches de durées unitaires suivies par une tâche de durée m. L'ordonnancement obtenu par l'Algorithme LSA est de longueur $2m$, d'autre part OPT=$m+1$.

Algorithme LPT (*Largest Processing Time*)

1. `Ordonner les tâches dans l'ordre décroissant de leur temps d'exécution` $p_1 \geq p_2 \geq \dots p_n$.
2. `Pour la liste obtenu utiliser l'Algorithme LSA.`

Théorème 1.2. *L'Algorithme LPT est un algorithme d'approximation avec un facteur 4/3 pour le problème d'ordonnancement des tâches.*

Preuve. `Cas 1 :` $p_n \leq \text{OPT}/3$. En raisonnant comme dans la preuve du Théorème 1.1, on déduit que

$$\text{TEMPS} \leq (\sum_{i=1}^{n-1} p_i)/m + p_n \leq \text{OPT} + \text{OPT}/3 = \frac{4}{3}\text{OPT}.$$

`Cas 2 :` $p_n > \text{OPT}/3$. Dans ce cas, chaque machine va exécuter au maximum 2 tâches et l'Algorithme LPT va retrouver l'ordonnancement optimal. $\square$

1.2. Voyageur de Commerce `(TSP : Traveling Salesman Problem)`

> Par une soirée d'hiver, au momemt où l'obscurité commençait à tomber, on aurait pu voir sur la route qui traverse le plateau de Marlborough une carriole, et dans cette carriole un homme qui pressait son cheval fatigué...
>
> *Histoire du commis voyager,* Ch. Dickens

Formulation : On se donne un graphe non orienté complet $G = (V, E)$, avec un coût entier positif ou nul $c(u, v)$ associé à chaque arête $(u, v) \in E$, et on doit trouver un circuit hamiltonien (un tour) dans G de coût minimum.

Théorème 1.3. *Si $P \neq NP$ et $\delta \geq 1$, il n'existe aucun algorithme d'approximation en temps polynomial avec un facteur δ pour TSP.*

Preuve. Supposons le contraire, donc pour un certain $\delta \geq 1$, il existe un tel algorithme $\mathcal{A}$. Nous allons montrer comment utiliser $\mathcal{A}$ pour résoudre en temps polynomial le problème CIRCUIT HAMILTONIEN. Soit $G' = (V, E')$ une instance de ce problème, $n := |V|$. On transforme G en une instance de TSP de la façon suivante : soit $G = (V, E)$ le graphe complet et on affecte un coût entier a chaque arête de E : $c(u, v) = 1$ si $(u, v) \in E$ et $c(u, v) = \delta \cdot n$, sinon. Évidemment, le tour optimal de G' est de coût n si et seulement si G possède un circuit hamiltonien. D'un autre côté, chaque tour avec une arête nouvelle est de coût $> \delta \cdot n$. Donc, un algorithme δ-approximatif doit retourner un circuit de coût n si il en existe un. $\square$

TSP Métrique : les coûts vérifient l'*inégalité triangulaire*

$$c(u,v) \leq c(u,w) + c(w,c) \text{ pour trois sommets } u,v,w \in V \text{ quelconques.}$$

Algorithme TSPM-Facteur2 (*TSPMétrique-Facteur 2*)

```
1. Construire un arbre couvrant minimal T pour G = (V,E,c).
2. Doubler chaque arête de T pour obtenir un graphe eulerien.
3. Trouver un circuit eulerien 𝒯 dans ce graphe.
4. Retourner le cycle hamiltonien 𝒞 de G qui visite les sommets de V
   dans l'ordre de leur première occurence dans 𝒯.
```

Théorème 1.4. *L'Algorithme TSPM-Facteur2 est un algorithme d'approximation avec un facteur 2 pour le problème de voyageur de commerce métrique.*

Preuve. Evidemment, coût(T)$\leq$ OPT (en supprimant une arête quelconque d'un tour on obtient un chemin, donc un arbre couvrant). Comme $\mathcal{T}$ contient deux copies de chaque arête de T, coût$(\mathcal{T})$ = 2$\cdot$coût(T). A cause de l'inégalité triangulaire coût$(\mathcal{C}) \leq$ coût$(\mathcal{T})$, donc coût$(\mathcal{C}) \leq 2\cdot$OPT. $\square$

Le pire exemple pout TSPM-Facteur2 est un graphe complet avec n sommets et des arêtes de coût 1 où 2 ($n = 6$). Il y a $2n - 2$ arêtes de coût 1 constituant un circuit de longeur $n - 1$ plus une étoile, le reste des arêtes ont toutes un coût 2. Supposons que TSPM-Facteur2 choisi l'étoile comme arbre couvrant et que le circuit eulerien $\mathcal{T}$ visite les sommets dans l'ordre 1234561. Dans ce cas le circuit $\mathcal{C}$ est de coût $2n - 2$ et OPT=n.

Algorithme TSPM-Facteur$\frac{3}{2}$ (*TSPMetrique-Facteur*$\frac{3}{2}$)

```
1. Construire un arbre couvrant minimal T pour G = (V,E,c).
2. Trouver un couplage parfait M de coût minimum dans le graphe complet
   induit par les sommets de degré impair de T. Ajouter M à T pour
   obtenir un graphe eulerien.
3. Trouver un circuit eulerien 𝒯 danc ce graphe.
4. Retourner le cycle hamiltonien 𝒞 de G qui visite les sommets de V
   dans l'ordre de leur première occurence dans 𝒯.
```

Théorème 1.5. *L'Algorithme TSPM-Facteur*$\frac{3}{2}$ *est un algorithme d'approximation avec un facteur 3/2 pour le problème de voyageur de commerce métrique.*

La démonstration est basée sur le résultat suivant.

Lemme 1.1. *Soit* $V' \subseteq V$ *un sous-ensemble de cardinalité paire, et soit* M *un couplage parfait de coût minimum de* V'. *Alors, coût*$(M) \leq$ *OPT/2.*

Preuve. Soit τ un tour optimal. Soit τ' le tour de V' obtenu en "court-circuitant" le tour τ. Par l'inégalité triangulaire, coût$(\tau') \leq$coût(τ). Le tour τ' est l'union de deux couplages parfaits, donc le couplage le moins coûteux est de coût$\leq$coût$(\tau')/2 \leq$ OPT/2. Alors, le couplage optimal est de coût $\leq$OPT/2. $\square$

Le pire exemple pour l'Algorithme TSPM-$\frac{3}{2}$ est le graphe suivant.

1.3. Couverture par des sommets (`Vertex Cover`)

Formulation : Une *couverture par des sommets* dans un graphe $G = (V, E)$ est un sous-ensemble $V' \subseteq V$ tel que si (u, v) est une arête de G, alors soit $u \in V'$, soit $v \in V'$ (ou les deux). Le problème de la *couverture par des sommets* consiste à trouver une couverture par de sommets de taille minimale.

Algorithme VC (*Minimum Vertex Cover*)

1. `Construire une couplage maximal` M `de` G `et retourner l'ensemble` V' `des extrémités des arêtes du couplage.`

Théorème 1.6. *L'Algorithme VC est un algorithme d'approximation avec un facteur 2 pour le problème de couverture par des sommets.*

Preuve. V' est une couverture par des sommets, sinon une arête non-couverte pourrait être ajoutée à M. Cependent, pour couvrir les arêtes de M, toute couverture par des sommets doit inclure au moins une extrémité de chaque arête de M, donc OPT$\geq |M|$. La couverture choisie par l'Algorithme VC est de cardinalité $2 \cdot |M|$, donc $|V'| \leq 2{\cdot}\text{OPT}$. □

Le pire exemple est le graphe biparti complet $K_{n,n}$. L'Algorithme VC retourne tous les $2n$ sommets de ce graphe et la couverture par des sommets optimale est de cardinalité n. Cela peut sembler étonnant, mais à ce jour, VC est l'algorithme d'approximation de couverture de sommet avec le meilleur facteur connu.

1.4. Couverture d'ensemble (`SC : Set Cover`)

Formulation : Soit U un ensemble fini de cardinalité n, et $S_1, \ldots, S_k$ une famille de sous-ensembles de U, chacun avec un coût non-négatif coût(S_i). Le problème consiste à trouver une sous-famille de coût minimum dont les membres couvrent U tout entier.

L'algorithme le plus simple est un algorithme glouton. Il choisit chaque fois l'ensemble le plus coût-efficace et supprime les éléments de U couverts par cet ensemble. Soit C l'ensemble des éléments déjà couverts. Dans l'itération courrante, la *coût-efficacité* d'un ensemble S et le coût moyen auquel S couvre les éléments non couverts, i.e., $\frac{\text{coût}(S)}{|S \cap \overline{C}|}$. Le *prix* d'un élément est le coût moyen auquel il a été couvert.

Algorithme SCGlouton (*Greedy Set Cover*)

1. $C := \emptyset$.
2. `while` $C \neq U$ `do`
   `Trouver l'ensemble le plus coût-efficace à cette itération, soit` S.
   `Soit` $\alpha = \frac{\text{coût}(S)}{|S \cap \overline{C}|}$, `i.e., la coût-efficacité de` S.
   `Prendre` S `dans la couverture et pour chaque` $e \in S - C$
   `faire prix`$(e) := \alpha$.
3. `Retourner les ensembles choisis.`

Soit $e_1, e_2, \ldots, e_n$ l'ordre dans lequel les éléments de U sont couverts.

Lemme 1.2. *Pour chaque* $k \in \{1, 2, \ldots, n\}$, *prix*$(e_k) \leq \frac{OPT}{n-k+1}$.

Preuve. Dans chaque itération, les éléments encore non-couverts peuvent être couverts par une sous-famille des ensembles restants de coût $\leq$OPT. Alors, il doit exister un ensemble ayant une coût-efficacité $\leq \frac{\text{OPT}}{|\overline{C}|}$ (un corollaire de la propriété suivante démontrée par induction : pour les nombres non-négatifs $a_1, \ldots, a_m$ et les nombres positifs $b_1, b_2, \ldots, b_m$, $\frac{\sum_{i=1}^m a_i}{\sum_{i=1}^m b_i} \geq \min_i \frac{a_i}{b_i}$). Dans l'itération dans laquelle l'élément e_k a été couvert l'ensemble $\overline{C}$ contient au moins $n - k + 1$ éléments. Comme e_k est couvert par l'ensemble le plus coût-efficace, on déduit que

$$\text{prix}(e_k) \leq \frac{\text{OPT}}{|\overline{C}|} \leq \frac{\text{OPT}}{n-k+1}. \qquad \square$$

Theorem 1.7. *L'Algorithme SCGlouton est un algorithme d'approximation avec un facteur $H_n \leq 1 + lgn$, où $H_n = 1 + \frac{1}{2} + \cdots + \frac{1}{n}$.*

Preuve. Comme le coût de chaque ensemble sélectionné est distribué entre les nouveaux éléments couverts par cet ensemble, le coût total de couverture est $\sum_{k=1}^{n} \text{prix}(e_k)$. Le lemme précédente implique que cette somme est $\leq (1 + \frac{1}{2} + \cdots + \frac{1}{n}) \cdot \text{OPT}$. □

Le pire exemple est constitué de $U = \{e_1, e_2, \ldots, e_n\}$ et $S_k = \{e_k\}$ avec $\text{coût}(S_k) = \frac{1}{k}$, $k = 1, \ldots, n$, et $S_{n+1} = \{e_1, e_2, \ldots, e_n\}$ avec $\text{coût}(S_{n+1}) = 1 + \epsilon$. L'algorithme va retourner la couverture $S_n, S_{n-1} \ldots, S_1$ de coût $H_n = \frac{1}{n} + \frac{1}{n-1} + \cdots + 1$, alors que le coût de la couverture optimale est de $1 + \epsilon$.

Cours II. La surséquence la plus courte (`Shortest superstring`)

> Ils demandèrent aux voisins quel est le mot le plus longue de la langue française. Une voisine qui avait des lettres répondit que c'était le mot *anticonstitutionnellement.*
>
> *La fée du robinet.* P. Gripari

Formulation : Soit Σ un alphabet fini et $S = \{s_1, \ldots, s_n\} \in \Sigma^*$ un ensemble de séquences (mots). Trouver la séquence la plus courte qui contient chaque s_i en tant que sous-séquence.

La *superposition* $\text{overlap}(s, t)$ de deux séquences s, t est le plus long suffixe de s qui est un préfixe de t. Soit $\text{prefix}(s, t)$ le préfixe de s obtenu en supprimant $\text{overlap}(s, t)$ de s. Soit $s \circ t$ la concaténation des séquences s et t (évidemment, $|s \circ t| = |s| + |t|$).

Algorithme SSGlouton

1. $T := S$.
2. `while` $|T| > 1$ `do`
   `Trouver dans` T `deux séquences` s, t `avec une superposition maximale et`
   `remplacer dans` T `les séquences` s, t `par prefix`$(s, t) \circ t$.

Conjecture : *L'Algorithme SSGlouton est un algorithme d'approximation avec un facteur de 2.*

Le pire exemple connu pour l'Algorithme SSGlouton est constitué des trois séquences $ab^k, b^k c$, et b^{k+1}. Si SSGlouton choisi en première itération les deux premières séquences, on obtient la surséquence $ab^k cb^{k+1}$; elle est deux fois plus longue que la surséquence optimale $ab^{k+1}c$.

2.1. PlusCourteSurséquence par CouvertureEnsemble

On peut supposer qu'aucune séquence s_i n'est une sous-séquence d'une autre séquence $s_j, j \neq i$. A partir de l'instance $S = \{s_1, \ldots, s_n\}$ de PlusCourteSurséquence on définit une instance $\mathcal{S}$ de CouvertureEnsemble de la façon suivante. Pour $s_i, s_j \in S$ et $k > 0$, si les k dernières lettres de s_i coincident avec les premières k lettres de s_j, soit σ_{ijk} la surséquence de s_i et s_j obtenu en identifiant cette partie commune de k lettres. Soit I la famille de toutes les séquences σ_{ijk}. L'instance $\mathcal{S}$ de CouvertureEnsemble est définit sur l'ensemble S et pour chaque $\pi \in S \cup I$ soit

$$\text{ensemble}(\pi) = \{s : s \in S : s \text{ est une sous-séquence de } \pi\}.$$

Le coût de π est $|\pi|$.

Algorithme SSviaSC(Shortest Superstring via Set Cover)

1. `Utiliser l'Algorithme SCGlouton pour l'instance` $\mathcal{S}$`. Soit`
   `ensemble`$(\pi_1), \ldots,$`ensemble`(π_k) `la couverture d'ensemble obtenue.`
2. `Returner` $s := \pi_1 \circ \pi_2 \circ \cdots \circ \pi_k$.

Soit $\text{OPT}_{\mathcal{S}}$ le coût d'une couverture optimale de S par les ensembles de $\mathcal{S}$.

Lemme 2.1. OPT$\leq$OPT$_{\mathcal{S}} \leq 2\cdot$OPT.

Preuve. À partir d'une solution optimale du problème de couverture on peut définir un surséquence en concaténant en ordre arbitraire les mots $\pi_1, \ldots, \pi_k$, donc OPT$\leq$OPT$_{\mathcal{S}}$.

Soit s une surséquence de longueur mimimale pour $s_1, \ldots, s_n$, donc $|s|$ =OPT. Supposons que les mots $s_1, \ldots, s_n$ sont numérotés dans l'ordre de leur première occurrence dans s. Pour $i < j$ la première occurrence de s_i doit se terminer avant la fin de l'occurence du mot s_j (sinon s_j serait une sous-séquence de s_i).

On va partitionner la liste $s_1, \ldots, s_n$ en groupes. Chaque groupe est une suite de mots consécutifs de cette liste ; on définit d_i et f_i les indices du premier et du dernier mot du groupe numéro i. Soit $d_1 = 1$ et soit f_i le plus grand indice d'un mot de notre suite qui intersecte le mot s_1. En général, si $f_i < n$, on définit $d_{i+1} = f_i + 1$ et soit f_{i+1} le plus grand indice d'un mot qui intersecte $s_{d_{i+1}}$. Éventuellement, on obtiendra $f_t = n$ pour un certain $t \leq n$.

Pour une paire de séquences (s_{d_i}, s_{f_i}), soit k_i la longueur de la superposition de leurs premières occurences dans s (elle pourrai être différente de overlap(s_{d_i}, s_{f_i})). Soit $\pi_i = \sigma_{d_i f_i k_i}$. Évidemment, $\{$ensemble$(\pi_i) : 1 \leq i \leq t\}$ est une solution admissible pour $\mathcal{S}$ de coût $\sum_{i=1}^{t} |\pi_i|$.

L'observation cruciale est que π_i et π_{i+2} sont disjointes. On va démontrer cette affirmation pour $i = 1$. Supposons le contraire, donc $\pi_1 \cap \pi_3 \neq \emptyset$. Donc l'occurence dans s de s_{d_3} intersecte l'occurence de s_{f_1}. Comme $f_1 < d_1 < d_3$, on déduit que s_{d_3} intersecte l'occurence de s_{d_2}. Mais dans ce cas, f_2 n'est pas le plus grand indice d'un mot qui intersecte s_{d_2}, une contradiction.

Cette observation implique que chaque lettre de s est couverte par au plus deux des π_i, alors OPT$_{\mathcal{S}} \leq \sum_{i}^{t} |\pi_i| \leq 2\cdot$OPT. □

Théorème 2.1. *L'Algorithme SSviaSC est un algorithme d'approximation avec un facteur de* $2 \cdot H_n$ *pour le problème SurséquencePlusCourte.*

2.2. PlusCourteSurséquence : une approximation avec un facteur 4

Quant aux mystiques, ils prétendent que l'extase leur révèle une chambre circulaire avec un grand livre également circulaire à dos continu, qui fait le tour complet des murs...

La Bibliothèque de Babel J.L. Borges

Soit s une surséquence optimale pour $s_1, \ldots, s_n$, donc sa longueur est OPT. Supposons que les mots $s_{i_1}, \ldots, s_{i_n}$ sont numérotés dans l'ordre de leur première occurrence dans s. On peut déduire que

$$\text{OPT} = |\text{prefix}(s_{i_1}, s_{i_2})| + |\text{prefix}(s_{i_2}, s_{i_3})| + \ldots + |\text{prefix}(s_{i_n}, s_{i_1})| + |\text{overlap}(s_{i_n}, s_{i_1})|.$$

Cette égalité établit un lien étroit entre la plus courte surséquence est le plus court tour du voyageur de commerce dans le *graphe-préfixe* $GP(S)$ de S. Les sommets de $GP(S)$ sont les entiérs $1, \ldots, n$, et pour $i \neq j$ il y a un arc $i \to j$ de poids $|\text{prefix}(s_i, s_j)|$ et un arc $j \to i$ de poids $|\text{prefix}(s_j, s_i)|$. Donc $|\text{prefix}(s_{i_1}, s_{i_2})| + |\text{prefix}(s_{i_2}, s_{i_3})| + \ldots + |\text{prefix}(s_{i_n}, s_{i_1})|$ est la longueur du tour $i_1 \to i_2 \to \ldots \to i_n \to i_1$. Par conséquence, la longueur du plus court tour dans le graphe-préfixe est une borne inférieure pour OPT.

Maintenant, au lieu des tours TSP on va considérer la Couverture de Circuit de Poids Minimum de $GP(S)$ (une *couverture de circuit* est une collection de circuits disjoints qui couvrent les sommets du graphe). Comme $i_1 \to i_2 \to \ldots \to i_n \to i_1$ est une couverture de circuit de $GP(S)$, alors la couverture de circuit de poids minimum de $GP(S)$ est aussi une borne inférieure pour OPT. Au contraire de TSP, la couverture de circuit de poids minimum peut être trouver en temps polynomial en utilisant l'algorithme de calcul d'un couplage parfait de poids minimum dans un graphe biparti $\Gamma = (U, V; E)$: soient $U = \{u_1, \ldots, u_n\}, V = \{v_1, \ldots, v_n\}$ et pour chaque couple $i, j \in \{1, \ldots, n\}, i \neq j$, on définit un arc $u_i \to v_j$ de poids $|\text{prefix}(s_i, s_j)|$. On peut facilement démontrer qu'un couplage parfait de Γ correspond à une couverture de circuit de $GP(S)$ de même poids et vice versa (un couplage de Γ correspond à une permutation de l'ensemble $1, \ldots n$ sans points fixes ; une telle permutation peut être écrite comme une union de circuits disjoints). A son tour, le problème de couplage parfait de poids minimum dans Γ peut être résolu en utilisant l'algorithme de flot de coût minimum.

Étant donné un circuit $c = (i_1 \to i_2 \to \dots i_l \to i_1)$ du graphe $GP(S)$, soit

$$\alpha(c) = \text{prefix}(s_{i_1}, s_{i_2}) \circ \dots \circ \text{prefix}(s_{i_{l-1}}, s_{i_l}) \circ \text{prefix}(s_{i_l}, s_{i_1}).$$

Notons que chacun des mots $s_{i_1}, \dots, s_{i_l}$ est une sous-séquence de la séquence infinie $(\alpha(c))^\infty := \alpha(c) \circ \alpha(c) \circ \dots$. Soit $\sigma(c) = \alpha(c) \circ s_{i_1}$. Alors $\sigma(c)$ est une surséquence pour $s_{i_1}, \dots, s_{i_l}$. Pour la suite, nous appelons s_{i_1} le *mot représentatif* de c.

Algorithme SS-Facteur4

1. `Construire le graphe-préfixe` $GP(S)$ `des séquences` $S = \{s_1, \dots, s_n\}$.
2. `Trouver une couverture de circuit` $\mathcal{C} = \{c_1, \dots, c_k\}$ `de poids minimum de` $GP(S)$.
3. `Retourner` $\sigma(c_1) \circ \dots \circ \sigma(c_k)$.

Lemme 2.2. *Si chaque séquence de $S' \subseteq S$ est une sous-séquence de t^∞ pour une séquence t, alors il y a un circuit de poids $|t|$ dans $GP(S)$ qui couvre les sommets correspondants aux séquences de S'.*

Preuve. Pour chaque séquence de S' on fixe le début de sa première occurrence dans t^∞. Comme aucun des mots de S n'est un sous-mot d'un autre mot de S, tous ces points de début sont deux à deux distincts et ils appartiennent tous à la première copie de t. Le circuit de $GP(S)$ qui visit les sommets correspondants dans cet ordre est de poids $\leq |t|$. □

Lemme 2.3. *Soient c et c' deux circuits de $\mathcal{C}$ et r, r' leur séquences représentatives, alors*

$$|overlap(r, r')| < poids(c) + poids(c').$$

Preuve. Supposons le contraire, donc $|\text{overlap}(r, r')| \geq \text{poids}(c) + \text{poids}(c')$. Soit α (resp., α') le préfixe de longueur poids(c) (resp., poids(c')) dans overlap(r, r').

Notons que overlap(r, r') est un préfixe de α^∞ et $(\alpha')^\infty$. De plus, α est un préfixe de $(\alpha')^\infty$ et α' est un préfixe de α^∞. Comme $|\text{overlap}(r, r')| \geq \text{poids}(c) + \text{poids}(c')$, on déduit qui α est α' commute, i.e. $\alpha \circ \alpha' = \alpha' \circ \alpha$. D'ici on obtient que $\alpha^\infty = (\alpha')^\infty$: pour $k > 0$ quelconque $\alpha^k \circ (\alpha')^k = (\alpha')^k \circ \alpha^k$, alors pour chaque $N > 0$ les préfixes de longueur N de α^∞ et de $(\alpha')^\infty$ coincident.

Le Lemme 2.2 implique que dans le graphe $GP(S)$ il existe un circuit de poids $\leq$ poids(c) qui couvre toutes les séquences de c et c', ce qui contredit l'hypotèse que $\mathcal{C}$ est une couverture de circuit de poids minimum. □

Théorème 2.2. *L'Algorithme SS-Facteur4 est un algorithme d'approximation avec un facteur 4.*

Preuve. Soit poids$(\mathcal{C}) = \sum_{i=1}^k \text{poids}(c_i)$. La surséquence retourné par L'Algorithme SS-Facteur4 est de longueur

$$\sum_{i=1}^{k} |\sigma(c_i)| = \text{poids}(\mathcal{C}) + \sum_{i=1}^{k} |r_i|,$$

où r_i est le mot représentatif du circuit c_i. On sait déja que poids$(\mathcal{C}) \leq$OPT. Ensuite, on affirme que la somme des longeurs des mots représentatifs est $\leq 3 \cdot$OPT.

Supposons que les $r_1, \dots, r_k$ sont numérotés dans l'ordre de leur première occurrence dans une surséquence optimale pour S. Le Lemme 2.3 implique que

$$\text{OPT} \geq \sum_{i=1}^{k} |r_i| - \sum_{i=1}^{k-1} |\text{overlap}(r_i, r_{i+1})| \geq \sum_{i=1}^{k} |r_i| - 2 \sum_{i=1}^{k} \text{poids}(c_i).$$

Donc,

$$\sum_{i=1}^{k} |r_i| \le \mathrm{OPT} + 2\sum_{i=1}^{k} \mathrm{poids}(c_i) \le 3 \cdot \mathrm{OPT}. \qquad \square$$

On peut améliorer l'Algorithme SS-Facteur4 pour obtenir une 3-approximation, mais la preuve de la correction n'est pas si simple.

Algorithme SS-Facteur3

1. Construire le graphe préfixe $GP(S)$ des séquences $S = \{s_1, \dots, s_n\}$.
2. Trouver une couverture de circuit $\mathcal{C} = \{c_1, \dots, c_k\}$ de poids minimum de $GP(S)$.
3. Éxecuter l'Algorithme SSGlouton sur $\{\sigma(c_1), \dots, \sigma(c_k)\}$ et retourner la surséquence résultante $\overline{\sigma}$.

Cours III. Sac à dos et remplissage de boîtes (Knapsack and Bin Packing)

3.1. Sac à dos (Knapsack)

Pour le petit déjeuner, George proposa d'empotrer des oeufs et du lard, ainsi que de la viandre froide, du thé, du pain, du beurre et de la confiture....,mais surtout pas de fromage! Le fromage, tout comme le pétrole, se montre volontiers envahissant. Il veut le bateau pour lui tout seul...

Trois hommes dans un bateau J.K. Jerome

Formulation : On se donne un ensemble $S = \{a_1, \dots, a_n\}$, chacun avec sa taille $size(a_i)$ et son profit $\mathrm{profit}(a_i)$ ainsi que la capacité B du sac à dos (tous les nombres sont des entiers positifs). On cherche à trouver un sous-ensemble d'objets de taille totale $\le B$ et de profit maximum.

Algorithme SacàDosGlouton

1. Trier les objets par l'ordre décroissant des rapports $\mathtt{profit}(a_i)/\mathtt{size}(a_i)$.
2. Trouver $P := max\{\mathtt{profit}(a_i) : i = 1, \dots, n\}$.
3. $Sac := \emptyset,\ b := B$.
4. for $i := 1$ to n do
   if $b \ge size(a_i)$ then
   $Sac := Sac \cup \{a_i\}$;
   $b := b - size(a_i)$.
5. Retourner $ProfitGlouton = max\{\mathtt{profit}(Sac), P\}$.

Théorème 3.1. *L'Algorithme SacàDosGlouton est un algorithme d'approximation avec un facteur 2.*

Preuve. Soit j l'indice du premier objet qui ne sera pas pris dans Sac. Le profit obtenu par l'algorithme jusqu'à ce point est $\mathrm{profit}_j = \sum_{i=1}^{j-1} \mathrm{profit}(a_i) \le \mathrm{profit}(Sac)$ et la taille est $size_j = \sum_{i=1}^{j-1} size(a_i) \le B$. On va montrer ensuite que $\mathrm{OPT} < \mathrm{profit}_j + \mathrm{profit}(a_j)$. Comme les objets sont ordonnés dans l'ordre décroissant en fonction du rapport profit/taille, si on échange un sous-ensemble d'objets $a_1, \dots, a_{j-1}$ avec un sous-ensemble admissible quelconque d'objets $a_j, a_{j+1}, \dots, a_n$, le profit global ne va pas augmenter. Donc, la valeur optimale de profit OPT est borné par profit_j plus le plus grand profit obtenu en remplissant l'espace libre restant (i.e., $B - size_j$) par des objets avec un rapport profit/taille $\le \mathrm{profit}(a_j)/size(a_j)$. Comme $size_j + size(a_j) > B$, on obtient

$$\mathrm{OPT} \le \mathrm{profit}_j + (B - size_j) \cdot (\mathrm{profit}(a_j)/size(a_j)) < \mathrm{profit}_j + \mathrm{profit}(a_j).$$

Si $\mathrm{profit}(a_j) \le \mathrm{profit}_j$, alors

$$\mathrm{OPT} \le 2 \cdot \mathrm{profit}_j \le 2 \cdot \mathrm{profit}(Sac).$$

Si profit$(a_j) >$ profit$_j$, alors $P >$ profit$_j$, donc

$$\text{OPT} \leq \text{profit}_j + \text{profit}(a_j) \leq \text{profit}_j + P < 2 \cdot \text{profit}_{max} \leq 2 \cdot ProfitGlouton. \qquad \square$$

3.2. FPAS et PTAS : des algorithmes pour les riches

Avant de continuer, quelques notions importantes.

Soit Π un problème d'optimisation NP-difficile ayant f_Π comme fonction objective. On dit que l'algorithme $\mathcal{A}$ est une *schéma d'approximation* pour Π si pour chaque instance I de Π et un paramètre d'erreur ϵ, la solution s retourné par $\mathcal{A}$ verifie

$$f_\Pi(I,s) \leq (1+\epsilon) \cdot \text{OPT}, \text{ si } \Pi \text{ est un problème de minimisation}$$

$$f_\Pi(I,s) \geq (1-\epsilon) \cdot \text{OPT}, \text{ si } \Pi \text{ est un problème de maximisation.}$$

$\mathcal{A}$ est un *schema d'approximation polynomial* (*polynomial time approximation scheme*, PTAS), si pour chaque $\epsilon > 0$ donné le temps d'exécution de $\mathcal{A}$ est polynomial dans la taille de l'instance I (mais peut être exponentiel en ϵ). Si le temps d'exécution de $\mathcal{A}$ est borné par un polynome de la taille de I et de $\frac{1}{\epsilon}$, alors $\mathcal{A}$ s'appelle un *schema d'approximation totalement polynomial* (*fully polynomial approximation scheme*-FPAS). C'est la meilleure chose que l'on peut esperér pour un problème NP-difficile.

La taille (binaire) $|I|$ d'une instance I d'un problème Π est le nombre de bits utilisés pour décrire les données numériques de I. La *taille unaire* $|I_u|$ de I est la taille nécessaire si on écrit toutes les données de I en système unaire (dans ce système, 10 s'écrit 1111111111). Un algorithme *pseudopolynomial* pour un problème Π est un algorithme dont le temps d'exécution pour toutes les instances I de Π est borné par un polynome en $|I_u|$.

On peut résoudre le problème SacàDos en temps pseudopolynomial en utilisant la programmation dynamique. Soit $P = max\{\text{profit}(a_i) : i = 1, \ldots, n\}$. Trivialement, nP est une borne supérieure pour une solution admissible. Pour $i \in \{1, \ldots, n\}$ et $p \in \{1, \ldots, nP\}$, soit $S_{i,p}$ un sous-ensemble de $\{a_1, \ldots, a_i\}$ dont le profit total est égal à p et la taille est minimum. Soit $A(i,p)$ la taille de $S_{i,p}$ ($A(i,p) = \infty$ si un tel sous-ensemble n'existe pas). On peut trouver de façon trivial les $A(1,p)$ pour $p \in \{1, \ldots, nP\}$. La récurrence suivante permet de calculer en temps $O(n^2P)$ tous les $A(i,p)$:

$$A(i+1,p) = min\{A(i,p), size(a_{i+1}) + A(i, p - \text{profit}(a_{i+1}))\}$$

si profit$(a_{i+1}) < p$ et $A(i+1,p) = A(i,p)$ sinon. Le profit maximal pour une taille bornée par B est $max\{p : A(n,p) \leq B\}$.

3.3. Un FPAS pour SacàDos

L'algorithme pseudo-polynomial précédent s'exécute en temps polynomial si tous les profits sont petits, i.e., si ils sont tous bornés par un polynome en n. L'idée derrière le FPAS pour SacàDos est de négliger un certain nombre de bits moins significatifs dans les profits des objets (leur nombre dépend de ϵ), en obtenant comme profits modifiés des nombres avec une écriture binaire polynomiale en n et $\frac{1}{\epsilon}$. L'algorithme précédent retournera une solution dont le profit et au moins $(1-\epsilon)$·OPT en temps polynomial en n et $\frac{1}{\epsilon}$.

Algorithme SacàDos-FPAS

1. Pour ϵ donnée, soit profit$'(a_i) = \lfloor$profit$(a_i)/K\rfloor$.
2. Avec les nouveaux profits des objets, en utilisant la programmation dynamique, trouver le sous-ensemble S' des objets les plus profitables.
3. Retourner le meilleur entre S' et l'objet de taille $\leq B$ avec un profit

`maximal` P.

Lemme 3.1. *Soit A l'ensemble choisi par SacàDos-FPAS . Alors, profit$(A) \geq (1-\epsilon) \cdot OPT$.*

Preve. Soit O l'ensemble optimal (de profit OPT). Pour chaque objet a_i on obtient $\text{profit}(a_i) \geq K \cdot \text{profit}'(a_i) \geq \text{profit}(a_i) - K$, donc

$$\text{profit}(O) - K \cdot \text{profit}'(O) \leq nK.$$

La programmation dynamique doit retourner un sous-ensemble dont le nouveau profit est au moins aussi grand qui le nouveau profit de O. Donc

$$\text{profit}(S') \geq K \cdot \text{profit}'(O) \geq \text{profit}(O) - nk = OPT - \epsilon P.$$

Comme l'algorithme prendre en compte l'objet le plus profitable, alors $\text{profit}(A) \geq P$. Donc,

$$\text{profit}(A) \geq \text{profit}(S') \geq \text{OPT} - \epsilon \cdot \text{profit}(A),$$

alors,

$$\text{profit}(A) \geq \frac{1}{1+\epsilon}\text{OPT}. \qquad \square$$

Théorème 3.2. *L'Algorithme SacàDos-FPAS est un FPAS pour sac à dos.*

Preuve. Le Lemma 3.1 implique que la solution retournée par SacàDos-FPAS est égal à OPT à un facteur $(1-\epsilon)$ près. Le temps d'exécution est $O(n^2\lfloor\frac{P}{K}\rfloor) = O(n^2\lfloor\frac{n}{\epsilon}\rfloor)$, et le résultat est démontré. $\square$

3.4. Remplissage des Boîtes (`Bin Packing`)

Formulation : On se donne un ensemble $E = \{e_1, \ldots, e_n\}$ des n objets, où la taille s_i de l'objet e_i vérifie $0 < s_i < 1$. On souhaite ranger tous les objets dans un minimum de boîtes de taille unitaire. Chaque boîte peut contenir tout sous-ensemble d'objets dont la taille globale n'excède pas 1.

Algorithme NextFit

`On prend chaque objet l'un après l'autre et on le place dans la dernière boîte si on peut, sinon l'objet est placé dans une nouvelle boîte.`

Théorème 3.3. *NextFit est un algorithme d'approximation avec un facteur 2.*

Preuve. Soit $S = \sum_{i=1}^{n} s_i$. On peut facilement voir que le nombre optimal de boîtes nécessaires est au moins $\lceil S \rceil$. D'autre part, NextFit utilise au maximum $2\lceil S \rceil$ boîtes : chaque paire de boîtes consécutives est remplie au moins à la moitié (donc de ≥ 1). $\square$

Le pire exemple est une instance de $4n$ objets $\{1/2, 1/2n, 1/2, 1/2n, \ldots, 1/2, 1/2n\}$, donc OPT$=n+1$, quand à NextFit, il utilise $2n$ boîtes.

Algorithme FirstFit (Fischer-Price)

`On prendre chaque objet l'un après l'autre et on le place dans la première boîte qui peut l'accueillir.`

On peut montrer que FirstFit est un algorithme d'approximation avec un facteur 1.7. Au lieu de cela, on va analyser un autre heuristique.

Algorithme FirstFitDecreasing

`Trier les objets dans l'ordre décroissant de tailles` $s_1 \geq s_2 \geq \ldots \geq s_n$ `et sur la liste résultante exécuter l'Algorithme FirstFit.`

Théorème 3.4. *FirstFitDecreasing est un algorithme d'approximation avec un facteur 1.5 (plus une boîte).*

Preuve. Supposons que la liste triée des objets $\{a_1, \ldots, a_n\}$ est partitionnée de la façon suivante :

$$A = \{a_i : s_i > 2/3\},$$

$$B = \{a_i : 2/3 \geq s_i > 1/2\},$$

$$C = \{a_i : 1/2 \geq s_i > 1/3\},$$

$$D = \{a_i : 1/3 \geq s_i\}.$$

Soit E' la solution retournée par FirstFitDecreasing. Si au maximum une boîte contient seulement des objets de groupe D, alors il existe une seule boîte (la dernière) remplie à moins de $2/3$, est la borne est prouvé. Dans le cas contraire, on affirme que FirstFitDecreasing trouve une solution optimale. Soit E_0 l'instance obtenu en supprimant de E tous les objets de groupe D. Dans ce cas, les valeurs trouvées par notre heuristique pour E et E_0 sont les mêmes. Donc il suffit de montrer l'optimalité de FirstFitDecreasing pour l'instance E_0. Pour cela, on observe que dans une solution admissible les objets de A sont seuls dans leurs boîtes et que chaque boîte contient au maximum deux objets (donc au maximum un de deux appartient au groupe B). FirstFitDecreasing va essayer de placer chaque objet de C ensemble avec le plus grand objet de B qui ne partage pas sa boîte avec un autre objet. Ça montre que le nombre de boîte utilisées par FirstFitDecreasing est égal à OPT. □

En effet, une analyse plus détaillée montre que le nombre de boîtes utilisées par FirstFitDecreasing est $\leq \frac{11}{9}\mathrm{OPT} + 4$.

On va continuer avec un algorithme de programmation dynamique pour RemplissagedesBoîtes où les n objets sont de k tailles différents $(i_1, \ldots, i_k)$ en spécifiant le nombre d'objets de chaque taille $(n_1, \ldots, n_k)$, $\sum_{i=1}^{k} n_i = n$. Soit $\mathrm{OPT}(i_1, \ldots, i_k)$ le nombre minimum de boîtes pour cette instance du problème. On commence par calculer tous les k-uples $(q_1, q_2, \ldots, q_k)$ tels que $\mathrm{OPT}(q_1, \ldots, q_k) = 1$ et $0 \leq q_i \leq n_i, 1 \leq i \leq k$. Après, on calcule tous les éléments du tableau k-dimensionel $\mathrm{OPT}(j_1, \ldots, j_k)$ pour chaque $(j_1, \ldots, j_k) \in \{0, \ldots, n_1\} \times \{0, \ldots, n_2\} \times \ldots \times \{0, \ldots, n_k\}$, en utilisant la formule récursive

$$\mathrm{OPT}(j_1, j_2, \ldots, j_k) = 1 + min\{\mathrm{OPT}(j_1 - q_1, \ldots, j_k - q_k) : (q_1, \ldots, q_k)\}.$$

Le calcul d'une valeur du ce tableau prend un temps $O(n^k)$, donc le tableau entier sera calculé en temps $O(n^{2k})$.

3.5. Un PTAS pour Makespan

Le problème d'Ordonnancement des Tâches (Makespan) est étroitement lié au problème de Remplissage des Boîtes : il existe un ordonnancement de n tâches de durées $p_1, \ldots, p_n$ sur m machines avec un makespan égal à t si et seulement si on peut ranger n objets de tailles $p_1, \ldots, p_n$ dans m boîtes de taille t chacune : $OPT = min\{t : \mathrm{OPT}(I, t) \leq m\}$. Ici I est l'instance de Bin Packing $\{p_1, \ldots, p_n\}$ et $\mathrm{OPT}(I, t)$ est le nombre minimum de boîtes de taille t pour ranger tous les objets. Dans le Cours I on a déjà noté que $LB = max\{\frac{1}{m}\sum_i p_i, max\{p_i\}\}$ est une borne inférieure pour OPT. D'autre part, l'Algorithme LSA montre que $2 \cdot LB$ est une borne supérieure pour OPT. Donc on pourra déterminer OPT par une recherche binaire dans l'interval $[LB, 2 \cdot LB]$, en résolvant à chaque itération un problème de Bin Packing par programmation dynamique. L'idée sera de tolérer deux types d'erreurs : l'erreur d'arrondi de certaines tailles pour obtenir un nombre borné de tailles et l'erreur due à la terminaison précoce de la recherche binaire pour assuré un nombre polynomial d'itérations.

Soit ϵ l'erreur donnée et soit $t \in [LB, 2 \cdot LB]$. Un objet sera dit *petit* si sa taille est $< t\epsilon$; les objets petits sont écartés pour le moment. Le reste des objets est arrondi de la façon suivante : chaque p_i dans l'interval $[t\epsilon(1+\epsilon)^i, t\epsilon(1+\epsilon)^{i+1})$ est remplacé par $p'_i = t\epsilon(1+\epsilon)^i$. Donc les p'_i résultants peuvent aprendre au maximum $k = \lceil log_{1+\epsilon}\frac{1}{\epsilon}\rceil$ valeurs distinctes. En appliquant la programmation dynamique aux objets arrondis on peut calculer le nombre minimum de boîtes de taille t pour ranger ces objets. Puisque l'opération d'arrondi réduit la taille de chaque objet avec un facteur $\leq 1+\epsilon$, si on retourne aux tailles initiales, on obtient un rangement admissible dans des boîtes de taille $t(1+\epsilon)$. Avec cette taille de boîtes, on va ranger les objets petits dans l'espace libre des boîtes en ouvrant des nouvelles boîtes seulement si c'est nécessaire. Donc, si une nouvelle boîte est ouverte, alors toutes les boîtes existantes contienent au moins t. Soit $\alpha(I,t,\epsilon)$ le nombre de boîtes utilisée par cet algorithme.

Lemme 3.1. $\alpha(I,t,\epsilon) \leq \mathrm{OPT}(I,t)$.

Preuve. Si l'algorithme n'ouvre pas de boîtes, le résultat est immédiat, puisque les objets arrondis sont rangés de façon optimale dans des boîtes de taille t. Sinon, toutes les boîtes (sauf la dernière) sont remplis à moins t, donc le remplissage optimal de I en boîtes de taille t doit utiliser au moins $\alpha(I,t,\epsilon)$ boîtes. □

La discussion précédente et le Lemme 3.1 impliquent que $\min\{t : \alpha(I,t,\epsilon) \leq m\} \leq$OPT. Donc, si la partie gauche est calculée de façon exacte, alors on obtient un makespan de durée $(1+\epsilon)$OPT. Maintenant, on va considérer l'erreur introduite par la recherche binaire dans $[LB, 2\cdot LB]$. À chaque itération, la taille du segment où la recherche binaire continue est réduit par un facteur de 2. Pour obtenir un segment de longeur $\epsilon \cdot LB$ on a besoin de $\lceil log_2\frac{1}{\epsilon}\rceil$ itérations. Soit T l'extrémité droite de ce segment. Donc $min\{t : \alpha(I,t,\epsilon) \leq m\}$ appartient au segment $[T-\epsilon\cdot LB, T]$. Donc $T \leq min\{t : \alpha(I,t,\epsilon) \leq m\} + \epsilon \cdot LB$. Comme $LB \leq$OPT et $min\{t : \alpha(I,t,\epsilon) \leq m\} \leq$OPT, on déduit que $T \leq (1+\epsilon)\cdot$OPT. Si $t = T$, alors on obtient un ordonnancement avec un makespan $\leq T\cdot(1+\epsilon)$. Finallement on est arrivé au résultat suivant.

Théorème 3.5. *L'algorithme produit un ordonnancement de durée* $\leq (1+\epsilon)^2\cdot$OPT.

Le temps d'exécution de cet algorithme est de $O(n^{2\lceil log_{1+\epsilon}\frac{1}{\epsilon}\rceil}\lceil log_2\frac{1}{\epsilon}\rceil)$.

Cours IV. Algorithmes en-ligne `(Online algorithms)`

Dans les problèmes d'optimisation considéres jusqu'ici toutes les données sont connues *a priori*. Les problèmes d'optimisation dont les données sont découverts au fur et à mesure (en-ligne) sont des *problèmes d'optimisation en-ligne.*

Par exemple, supposons que nous soyons dans dans une station de ski où on peut louer une pair de ski pour 30F la journée ou on peut acheter une paire pour 300F. On restera tant qu'il fait beau, donc nous ne connaissons pas notre jour de départ. Quelle doit être notre stratégie pour avoir le moins de dépences par rapport à une personne (notre adversaire) qui connais exactement la fin du beau temps (disons le directeur du Meteo France) ? Une autre version : combien de fois un skieur débutant doit louer des skis avant de s'acheter une paire ? On peut proposer une méthode simple qui dans le pire des cas nous fait dépanser deux fois plus qui notre l'adversaire : il faut louer le premièrs 9 jours, après il faut acheter (une méthode de ce type s'appele “first rent, then buy”) . On dit que ce algorithme est 2-compétitif.

De façon générale et précise, si Π est un problème d'optimisation en-ligne et ALG est un algorithme en-ligne pour Π, on dit que ALG est *c-compétitif* si il existe une constante α telle que pour chaque instance $I \in \Pi$,

$$\text{coût}_{\text{ALG}}(I) \leq c \cdot \text{coût}_{\text{OPT}}(I) + \alpha,$$

où OPT est l'algorithme optimal off-line (qui connait I a priori).

4.1. Pagination à la demande (`Paging problem`)

Ne remets jamais au lendemain ce que tu peux faire le surlendemain
Alphonse Allais

Formulation : On va considérer un système de mémoire virtuelle à deux niveaux (virtual memory system). Chaque niveau peut garder un certain nombre de pages (unités de mémoire d'une taille fixe). Le premier niveau (*la mémoire secondaire*- `slow memory`) est constitué d'un grande nombre N de pages $P = \{p_1, \dots, p_N\}$, quand au deuxième niveau (la *mémoire vive*- `fast memory`), elle peut accueillir a un instant maximum k pages de P ($k << N$). À la requête d'une page $p_i \in P$, le système doit rendre cette page disponible dans la mémoire vive. Si p_i se trouve déjà là, le système ne fait rien. Sinon, le système doit subir un *défaut de page (page fault)*, donc il doit transférer p_i dans la mémoire vive à la place d'une page résidant là. Pour cela, le système doit décider quelle page sera expulser de la mémoire vive (la page *victime*). Le but est de minimiser le nombre de défauts de pages (*taux de défaut de page*).

A cause de l'importance de ce problème, quelques stratégies sont connues/utilisées :
LRU (Least Recently Used) : Remplacer la page la moins récemment utilisée.
FIFO (First In First Out) : Remplacer la page qui est restée dans la mémoire vive le plus longtemps (la page la plus ancienne).
LIFO (Last In First Out) : Remplacer la page que a eté placé dans la mémoire vive en dernier.
FWF (Flush When Full) Si il y a un défaut de page, expulser toutes les pages résidant dans la mémoire vive.
LFD (Longest Forward Distance) : Remplacer la page dont la demande suivante sera la plus tardive.

LFD est un algorithme off-line, puisque il utilise l'information sur les requêtes prochaines, tous les autres sont des algorithms en-ligne.

Théorème 4.1. *L'algorithme LFD est un algorithme optimale off-line pour le problème de pagination.*

Preuve. On va montrer que n'importe quel algorithme optimal de pagination peut être transformer en LFD sans diminuer ses performances. Notamment, on va démontrer la proprieté suivante :

Lemme 4.1. *Soit ALG un algorithme off-line de pagination et soit σ une suite de requêtes. Pour chaque $i = 1, 2, \dots, |\sigma|$, il est possible de construire une suite ALG_i d'algorithmes off-line telles que :*
(i) ALG_i va exécuter les premiers $i-1$ requêtes exactement comme ALG;
(ii) si la requête i est une défaut de page, alors ALG_i remplace la page dont la demande suivante est la plus tardive.
(iii) $\text{coût}_{ALG_i}(\sigma) \leq \text{coût}_{ALG}(\sigma)$.

Preuve de Lemme 4.1. Pour un ensemble de pages X et une page p, soit $X + p$ l'union de X et de $\{p\}$. Supposons que immédiatement après la requête numéro i les mémoires vives de ALG et ALG_i contiennent les pages $X + v$ et $X + u$ respectivement, où X est un ensemble de $k-1$ pages et u, v sont deux pages quelconques. On peut supposer sans perte de généralité que $u \neq v$, donc la requête i est un défaut de page. Avant que v ne soit demandé, ALG_i va servir les requêtes exactement comme ALG, sauf qui il va expulser u si ALG expulse v. Le nombre de pages communes est $k-1$. Maintenant, si après une requête ce nombre devient k (donc ALG expulse la page v), à partir de ce moment ALG_i est identifié à ALG.

Si la page v est demandée, mais ALG et ALG_i ne sont pas encore identique, cela va attirer un défaut de page pour ALG_i est pas pour ALG. Comme v au moment de son expulsion par ALG_i était la page dont la demande suivante était la plus tardive et à ce moment u était aussi dans la mémoire vive, donc entre l'expulsion de v par ALG_i et cette requête de v on doit avoir au moins une requête de la page u. A la première requête de u, ALG fera une défaut de page et ALG_i non. Donc les nombres des défauts de page de ALG et ALG_i juste après le service de page v sont égaux. Maintenent, pour servir v, ALG_i va expulser la page u, donc à partir de ce moment ALG et ALG_i ont k pages en commun. □

Pour démontrer le théorème, on va utiliser $n = |\sigma|$ fois le Lemme 4.1. Soit OPT un algorithme optimal. On utilise le lemme pour obtenir OPT_1 à partir de OPT, après pour obtenir OPT_2 à partir de OPT_1, etc. Alors OPT_n coincide avec LRU et (iii) implique que

$$\text{coût}_{\mathrm{OPT}}(\sigma) = \text{coût}_{\mathrm{OPT}_1}(\sigma) = \text{coût}_{\mathrm{OPT}_2}(\sigma) = \ldots = \text{coût}_{\mathrm{OPT}_n}(\sigma) = \text{coût}_{\mathrm{LRU}}(\sigma) \qquad \square$$

On va continuer avec une série d'algorithmes de marquage. Soit σ une suite de requetês. On divise cette suite en *phases* de la façon suivante. La phase 0 est vide. Pour chaque $i \geq 1$, la phase i est la séquence maximale de requêtes qui suit la phase $i-1$ et qui contient des requêtes au maximum de k pages distincts. Donc, la phase $i+1$ va commencer par la requête de la $(k+1)$ème page différente après le début de la phase i. Une telle partition s'appele une *partition en k-phases.* On va affecter à chaque page de mémoire secondaire un bit appelé la *marque* de cette page. Si le bit d'une page est 1 on dit que la page est *marquée*, sinon elle est *nonmarquée.* Supposons qui au début d'une phase i toutes les pages dans la memoire vive deviennent nonmarqués. Pendant la phase i nous marquerons une page quand elle sera demander pour la première fois dans la mémoire vive. Un *algorithme de marquage* n'expulse jamais une page marqué de la mémoire vive. L'Algorithme FWF est un exemple d'algorithme de marquage.

Théorème 4.2. *Soit ALG un algorithme quelconque de marquage. Alors ALG est un algorithme k-compétitif de pagination.*

Preuve. Soit σ une suite de requêtes partitionnée en k-phases. Nous affirmons que à chaque phase $i \geq 1$, l'algorithme ALG subira au maximum k défauts de pages. En effet, à chaque phase les requêtes concernent k pages différentes (au maximum k pages dans la dernière phase). Si la page a été demandée elle est marquée et elle ne pourra pas être expulsée jusqu'à la phase suivante. Donc ALG ne peut pas faire deux défauts de pages sur la même page.

Pour $i \geq 1$, soit q la première requête de phase i. On va considérer la suite σ_0 de requêtes qui commence par la sèconde requête de la phase i et fini par la première requête de la phase $i+1$ (en supposant qu'une telle phase existe). Nous affirmons qu'un algorithme optimal off-line OPT quelconque fera au minimum un défaut de page sur cette suite de requêtes. OPT contient la page q et $k-1$ autres $k-1$ pages. Si OPT ne fait pas de défauts pendant la suite σ_0, il doit garder toutes les k pages demandées pendant la phase i dans la mémoire vive. La première requête de la phase $i+1$ (donc la dernière requête de σ_0) est une nouvelle page. Donc OPT fera un défaut de page. $\square$

Théorème 4.3. *L'Algorithme LRU est un algorithme de marquage, donc il est k-compétitif.*

Preuve. Soit σ une suite de requêtes partagée en k-phases. Supposons par contradiction, que LRU n'est pas un algorithme de marquage. Donc il va expulser une page x pendant une phase i. On va considérer la première requête de x pendant cette phase i. Après que la requête de x a été servi, x devient la page de la mémoire vive la plus récemment utilisée. Au moment où x est expulsée par LRU, la mémoire vive de LRU est pleine (donc elle contient k pages). Par définition de LRU, la page x est expulsée parce qu'elle est la page la moins récemment utilisée résidant dans la mémoire vive. Donc pendant la phase i il y a une requête de page x, il y a des requêtes des autres $k-1$ pages et, finalement, une requête de page que va remplacer x, au total $k+1$ pages distincts, ce qui contredit la définition d'une k-phase. $\square$

Théorème 4.4. *Soit ALG un algorithme de pagination. Alors sa compétitivité est $\geq k$. En particulier, LRU est un algorithme de pagination en-ligne optimal.*

Preuve. On va utiliser le résultat suivant.

Lemme 4.2. *Pour une suite de requêtes σ d'un ensemble de $k+1$ pages, l'Algorithme LFD fait au maximum $\frac{|\sigma|}{k}$ défauts de pages.*

Preuve du Lemme 4.2. Supposons que après le service de la requête numéro j, LFD expulse la page p. Par définition de LFD et comme il y a seulement $k+1$ pages dans la mémoire secondaire, toutes les autres pages de la mémoire vive doit être démandées avant la prochaine requête de la page p. Donc LFD fait un seul défaut sur une liste de k requêtes. □

Pour en finir avec le Théorème 4.4, supposons que la mémoire secondaire posséde seulement $k+1$ pages $p_1, \ldots, p_k, p_{k+1}$. On va présenter une suite de requêtes longue et cruelle σ pour laquelle ALG fera $|\sigma|$ défauts de pages (alors le Lemme 4.2 va impliquer que la compétitivité de ALG est $\geq k$). Supposons que ALG contient les pages $p_1, \ldots, p_k$ dans sa mémoire vive. La séquence cruelle σ est défini de façon inductive : $r_1 = p_{k+1}$ et r_{i+1} est la seule page qui n'est pas située dans la mémoire vive après que ALG ait servi les requêtes $r_1, r_2, \ldots, r_i$. Cette séquence σ est bien défini puisque ALG est déterministe. D'autre part, σ peut être avoir une longueur arbitraire et ALG commet un défaut de page à chaque requête. □

4.2. L'analyse de compétitivité par des fonctions potentiel

La méthode du potentiel fonctionne comme suit. Un algorithme en-ligne va exécuter une suite de requêtes, donc il va effectuer une suite d'opérations. Si il y a deux telles suites, une issue de l'algorithme ALG et l'autre issue de l'algorithme OPT, on peut combiner les deux suites en ordre chronologique dans une seule longue suite. Une partition de cette suite s'appelle une *séquence d'événements* est chaque segment s'appelle un *événement* (par exemple, les opérations effectuées par LRU et LFD pendant une phase). La *configuration* d'un algorithme est son état par rapport à l'extérieur (pour un algorithme de pagination, c'est les pages contenues dans la mémoire vive). Notons par S_{ALG} et S_{OPT} toutes les configurations possibles des algorithmes ALG et OPT. Soit Φ une application $\Phi : S_{\text{ALG}} \times S_{\text{OPT}} \to R$. Autrement dit, Φ associe a chaque couple des configurations de ALG et OPT un nombre réel. Chaque fonction de ce type s'appele une *fonction potentiel.* L'idée sera de définir des fonctions potentiels qui vérifient certains conditions spécifiques.

Soit $e_1, e_2, \ldots, e_n$ une séquence d'événements pour une suite σ des requêtes. Pour $i = 1, \ldots, n$, soit Φ_i la valeur de Φ après l'événement i. Soit Φ_0 la valeur de Φ just après le début, donc Φ_0 est une constante qui dépend seulement des configurations initiales de ALG et OPT. Supposons que nous voulons démontrer que ALG est c-compétitif. Deux approches via les fonctions potentiel sont disponibles.

Coûts amortis : Soient coût$_{\text{ALG}_i}$ et coût$_{\text{OPT}_i}$ les coûts de l'événement numéro i de l'algorithmes ALG et OPT (les *coûts réels* de l'événement i). Soit Φ une fonction potentiel. Pour chaque événement e_i le *coût amorti* a_i de e_i est

$$a_i = \text{coût}_{\text{ALG}_i} + \Phi_i - \Phi_{i-1}.$$

Pour démontrer que ALG est c-compétitif, il suffit de vérifier les conditions suivantes pour une suite $e_1, \ldots, e_n$ quelconque :

1) pour chaque événement e_i, $a_i \leq c \cdot \text{coût}_{\text{OPT}_i}$;
2) il existe une constante b indépendante de σ telle que $\Phi_i \geq b$ pour tout i.

D'après la définition du coût amorti, le coût amorti total des n événements est

$$\sum_{i=1}^{n} a_i = \sum_{i=1}^{n} (\text{coût}_{\text{ALG}_i} + \Phi_i - \Phi_{i-1}) = \sum_{i=1}^{n} \text{coût}_{\text{ALG}_i} + \Phi_n - \Phi_0 = \text{coût}_{\text{ALG}} + \Phi_n - \Phi_0.$$

(la seconde égalité se déduit facilement, puisque les Φ_i se superposent). D'autre part,

$$\sum_{i=1}^{n} a_i \leq \sum_{i=1}^{n} c \cdot \text{coût}_{\text{OPT}_i} = c \cdot \text{coût}_O PT.$$

Donc

$$\text{coût}_{\text{ALG}} \leq c \cdot \text{coût}_{\text{OPT}} + \Phi_0 - \Phi_n.$$

Mouvements alternée : (sa justification est la même que cella de la méthode précédente) Pour prouver la c-compétitivité de ALG il suffit de trouver une fonction potentiel, tel que :

1) si seul l'adversaire OPT se déplace pendant l'événement e_i et il paye x pour ce mouvement, alors $\Delta\Phi = \Phi_i - \Phi_{i-1} \leq cx$;

2) si seul ALG se déplace pendant l'événement e_i et il paye x pour ce mouvement, alors $\Delta\Phi = \Phi_i - \Phi_{i-1} \leq -x$.

3) il existe une constante b indépendante de la suite σ tel que $\Phi_i \geq b$ pout tous i.

4.3. Le problème des k-serveurs (k-`server problem`)

> `Le marchand donna donc cent pièces d'or à chacun de ses fils, et les voilà partis tous les trois pour la ville afin d'acheter des marchandises. Le fis aîne, qui s'est levé tôt, arrive le premier. Le second fils arrive ensuite, et charge son bateau d'un cargaison de miel. Quant à Manque-de-Chance, il se lève sur le coup de midi, déjeune sans se presser, et se met en route vers deux heures. Mais avant d'arriver à la ville, il rencontre en chemin une bande d'enfants qui ont attrapé un chat...`
>
> *Je-ne-sais-qui, je-ne-sais-quoi*, P. Gripari

Formulation. Soit $k \geq 1$ un entier et soit (M, d) un espace métrique (donc (a) $d(x, y) \geq 0$ et $d(x, y) = 0$ ssi $x = y$ et (b) $d(x, y) \leq d(x, z) + d(z, y)$ pour $x, y, z \in M$) avec au moins $k + 1$ points. L'algorithme ALG controle k serveurs mobiles positionnés dans des points de l'espace M. La séquence des requêtes $\sigma = r_1, r_2, \ldots, r_n$, représente une suite de points r_i de M. Une requête r_i est *servie* si un certain serveur se trouve dans le point r_i. En déplaçant les serveurs, l'algorithme doit desservir les requêtes au fur et à mesure. Pour une séquence σ, le coût de ALG est la distance globale parcourus par ces serveurs pour déservir la séquence de requêtes σ. Le probléme des k serveurs est de trouver des algorithmes en-lignes compétitifs.

La conjecture des k-serveurs. *Pour n'importe quel espace métrique (M, d) il existe un algorithme en-ligne k-compéritif pour le problème des k-serveurs.*

Après des efforts considérables, cette conjecture est "presque" résolu : il existe un algorithme qui est $(2k - 1)$-compétitif pour n'importe quel espace métrique. D'autre part, la conjecture a été verifié pour la droite et les arbres et toutes les valeurs de k, le plan avec la norme l_1 et $k = 3$, ainsi que pour $k = 2$ et tous les espaces métriques. Dans ce cours, on se contentera d'analyser le problème des k-serveurs sur la droite et les arbres, aussi que le cas $k = 2$ dans un espace euclidien.

L'inegalité triangulaire implique qu' à chaque requête r_i il suffit de déplacer un seul serveur (un algorithme de ce type est dit *paresseux*).

L'idée la plus simple est d'employer un algorithme GLOUTON : *déservir la requête r_i par le serveur le plus proche du point r_i*. On peut facilement établir que l'algorithme GLOUTON n'est pas compétitif. Pour cela, soit $k = 2$ et soit $M = \{a, b, c\}$, $d(a, b) = 1, d(b, c) = 2$, et $d(a, c) = 3$. Pour n'importe quel positionnement initial des deux serveurs du GLOUTON, il va déservir la séquence des requêtes $c, b, a, b, a, b, a, b, a, b, \ldots$ de façon suivant : un serveur va rester dans le point c pour "l'éternité", et toutes les requêtes a et b seront desservies par le deuxième serveur, donc le coût sera nonborné. Par ailleurs, l'algorithme optimal off-line va desservir les requêtes avec un coût de $d(a, b) + 2d(b, c) = 5$.

Pagination-un cas particulièr de k-serveurs : M est l'ensemble des N pages, $d(x, y) = 1$ pour $x \neq y$ et les positions des k serveurs correspondent aux k pages résidants dans la mémoire vive. En d'autres termes, le problème de pagination est exactement le problème des k-serveurs dans un graphe complet.

Théorème 4.5. *Soit (M, d) un espace métrique avec au moins $k+1$ points. N'importe quel algorithme en-ligne pour le problème des k-serveurs en (M, d) est de compétitivité $\geq k$.*
(sans preuve)

Un algorithme k-compétitif pour la droite et les arbres

Algorithme CD-LIGNE (*CouvertureDouble-Ligne*)

> Si la requête r_i n'appartient pas à l'enveloppe convexe des serveurs, alors desservir r_i par le serveur le plus proche de r_i. Sinon, r_i appartient au segment entre deux serveurs voisins. Alors, déplacer ces serveurs à une vitesse uniforme jusqu'à ce qu'un serveur arrive au point r_i.

Exercice. Analyser les actions de CD-LIGNE pour l'exemple où GLOUTON a échoué.

Théorème 4.6. *L'Algorithme CD-LIGNE est k-compétitif.*

Preuve. Soit M_{min} le couplage de coût minimum entre les serveurs de CD-LIGNE et OPT, un algorithme off-line optimal (OPT est un algorithme paresseux). Soit σ_{CD} la somme des distances entre les serveurs du CD-LIGNE : $\Sigma_{\text{CD}} = \sum_{i<j} d(s_i, s_j)$. La preuve utilise la fonction potentiel suivante :

$$\Phi = k \cdot M_{min} + \sigma_{\text{CD}}.$$

Elle est non-négative, donc elle possède une borne inférieure. Par la deuxième méthode, il suffit de démontrer que (i) si OPT déplace ses serveurs de δ unités, alors le potentiel croît de *maximum* $\leq k\delta$ unités et (ii) si CD-LIGNE déplace ses serveurs de δ unités, alors le potentiel decroît de *minimum* $\geq \delta$ unités.

L'affirmation (i) est triviale : Σ_{CD} reste inchangé, quand à M_{min}, sa valeur peut augmenter de maximum δ unités, puisque OPT va déplacer un seul serveur. Pour (ii) on va distinguer deux cas.

Si CD-LIGNE déplace un seul serveur, alors ce serveur est une extrémité de l'enveloppe convexe des serveurs. Donc le serveur va s'éloigner des $k-1$ autres serveurs, donc $\sum_{\text{CD}}$ va augmenter de $(k-1)\delta$ unités. D'autre part, on peut facilement voir qu'il existe un couplage optimal dans lequel le serveur mobile sera en couple avec le serveur de OPT qui sert la requête courrante r_i. Alors M_{min} décroît d'au moins δ unités. Donc Φ décroît d'au moins $k\delta-(k-1)\delta = \delta$ unités, c'est à dire exactement le déplacement effectué par CD-LIGNE.

Maintenant, supposons que CD-LIGNE déplace deux serveurs s_1, s_2, donc $r_i \in [s_1, s_2]$. Dans un couplage optimal le serveur de OPT qui est dans le point r_i sera en couple avec un de ces deux serveurs. L'autre serveur mobile peut s'éloigner de maximum δ unités de son partenaire, donc globalement M_{min} ne va pas augmenter. Pour analyser comment Σ_{CD} va changer, notons que la somme des distances de s_1 et s_2 jusqu'à un autre serveur s de CD-LIGNE ne va pas changer : un des deux serveurs va s'approcher de s de δ unités et l'autre va s'éloigner de s de δ unités. Comme $d(s_1, s_2)$ décroît de 2δ, le potentiel va décroître de 2δ unités, et 2δ est le déplacement effectué par l'algorithme. □

Maintenant, supposons que $T = (M, d)$ est un arbre. Chaque arête (x, y) est un segment de droite de longueur $d(x, y)$. Étant donné une configuration des k serveurs et une requête r, le serveur s_j est *voisin* de r si le (seul) chemin de T entre s_j et r ne contient pas d'autres serveurs.

Algorithme CD-ARBRES (*CouvertureDouble-ARBRE*)

> Étant donnée une requête r_i, déplacer vers r_i de façon continu et à vitesse constante tous les serveurs voisins de r_i, jusqu'à ce qu'un serveur arrive au point r_i.

Pendant ce mouvement, certains serveurs mobiles au début peuvent être bloqués par d'autres serveurs, donc ils doivent s'arrêter.

Théorème 4.7. *L'Algoritme CD-ARBRE est k-compétitif.*

Preuve. La preuve va généraliser celle du Théorème 4.6. L'affirmation (i) est analogue. Pour analyser la différence de potentiel quand CD-ARBRES va déplacer ses serveurs, on va partager les actions de cet algorithme en phases : pendant une phase l'ensemble des serveurs voisins (donc des serveurs qui vont se déplacer) est fixe. Considerons une phase où il y a m serveurs voisins et supposons que chacun des m serveurs se déplace de δ unités (donc $m\delta$ unités au total). Comme dans la preuve précédente, il existe un couplage minimal dont l' un des m serveurs sera en couple avec le serveur adverse positionné en r_i, donc M_{min} va diminuer de δ. D'autre part, dans le pire des cas les autres $m-1$ serveurs vont s'éloigner de leurs partenairzs, donc ça va augmenter M_{min} de maximum $(m-1)\delta$. Au total, l'echange dans le couplage minimum est de maximum

$$\Delta M_{min} \leq (m-2)k\delta = mk\delta - 2k\delta.$$

Pour analyser l'échange dans la composante Σ_{CD}, soit s un serveur immobile pendant cette phase. Un des serveurs mobiles va s'éloigner de δ unités de s, quand aux autres $m-1$ serveurs mobiles, chacun va parcourir une distance de δ unités vers s. En prenant la somme sur tous les $k-m$ serveurs immobiles, ça va entrainer une diminution de

$$(k-m)(m-2)\delta = km\delta - 2k\delta - m^2\delta + 2m\delta$$

de la fonction potentiel. Finalement, chaque paire de serveurs mobiles va s'approcher de 2δ unités. En prenant la somme sur toutes les $\frac{m(m-1)}{2}$ paires, on obtient une diminution supplémentaire de $\delta m(m-1) = \delta m^2 - \delta m$. Donc Σ_{CD} va decroître de $km\delta - 2k\delta + m\delta$. Alors

$$\Delta\Phi \leq (mk\delta - 2k\delta) - (mk\delta - 2k\delta + m\delta) = -m\delta.$$

Mais $m\delta$ est exactement la distance parcourue par les serveurs de CD-ARBRES pendant cette phase. $\square$

Le résultat précédent a une application pour le problème de pagination. On peut plonger le graphe complet sur N sommets dans une étoile S avec N feuilles $p_1, p_2, \ldots, p_N$ plus un sommet central v. La longueur de chaque arête de cette etoile est de $\frac{1}{2}$. Le sommet p_i représente la page i dans la memoire secondaire. Un déplacement d'un serveur d'une page à l'autre a un coût unitaire, i.e. le coût d'un défaut de page. Donc un algorithme pour les k serveurs dans cette étoile (plus précisément, sa version parésseux) peut servir comme algorithme de pagination.

L'Algorithme CD-ARBRE sur l'étoile S avec des requêtes dans des feuilles fonctionne de la façon suivante. Soit r_i une nouvelle requête. Si il y a un serveur en r_i, la requête sera desservie par ce serveur. Dans le cas contraire, si au moins un serveur est placé en v, alors un de ces serveurs va desservir r_i. Finalement, si tous les k serveurs sont dans des feuilles de S distinctes de r_i, CD-ARBRE va transferer tous ses k serveurs au sommet central v et il va desservir r_i avec un serveur quelconque. Tous les autres $k-1$ serveurs vont rester dans v. Chacune des $k-1$ requêtes suivantes aux pages qui sont pas en mémoire vive, seront desservies par des serveurs positionnés en v. Donc, dans ce cas CD-ARBRES est equivalent avec l'algorithme FWF (Flush When Full) de pagination.

2-Serveurs dans un espace euclidien

Notons par x, y les serveurs. Pour un point r de E^n soit

$$\alpha(x,y,r) = d(x,y) + d(x,r) - d(y,r),$$

où $d(\cdot)$ est la distance euclidiene.

Algorithme CD-EUCLIDIEN (*CouvertureDouble-Ligne*)

> Si r_i est la requête courrante et x est le serveur le plus proche de r_i, alors déplacer le serveur y de $\frac{1}{2}\alpha(x,y,r_i)$ vers x et après desservir r_i par le serveur x.

Théorème 4.8. *L'Algorithme CD-EUCLIDIEN est* 3*-compétitif.*

4.4. Le problème de fichier migrant (`File migration problem`)

1. Ne jamais attaquer tout de suite. Se faire remarquer de loin, plusieurs jours auparavant, en émettant des signaux de fumée bien visibles...
3. Laisser des traces évidantes de son passage : empreintes de chevaux, feux de bivouac éteintes, plumes...
28. En bondissant sur le dos d'un Blanc, tenir son propre couteau de façon à ne pas le blesser... Attendre que le Blanc se retourne.

Comment être un Indien, U. Eco

Formulation. Soit $\mathcal{N}$ un réseau de processeurs (par exemple, le web), un fichier F est situé dans un noeud de $\mathcal{N}$ (disons, F est une base de données). Il y a un suite de requêtes en-ligne à F à partir de certains processeurs de $\mathcal{N}$. Le coût de chaque requête est la distance en $\mathcal{N}$ entre le processeur-demandeur et le processeur accueillant le fichier F. En plus de ce type de requêtes, l'algorithme de temps en temps peut migrer F d'un processeur à l'autre avec un coût égal à la distance entre les processeurs multiplier par la taille D du fichier F. Le coût total de l'algorithme est la somme des coûts d'accès à F plus le coût de déplacement. Le problème de fichier migrant est de trouver des algorithmes en-lignes compétitif.

Jusqu'à ce jour, il existe un réseau avec 4 sommets pour lequel n'importe quel algorithme en-ligne est de compétitivité $\geq \frac{85}{27}$ =3,148. D'autre part, il existe un algorithme pour tous les réseaux avec une compétitivité de 4,086. Cet algorithme est un raffinement de l'algorithme suivant.

Algorithme MTM (*Move To Min*)

> `L'algorithme divise la suite des requêtes en phases. Chaque phase consiste en` D `accès consécutifs des processeurs` $v_1, \ldots, v_D$. `Pendant une phase l'algorithme ne déplace pas de fichier. A la fin d'une phase le fichier sera transférer au processeur` m `qui minimise la somme des distances jusqu'à` $v_1, \ldots, v_D$.

Théorème 4.9. *L'Algoritme MTM est* 7*-compétitif pour n'importe quel réseau* $\mathcal{N}$.

Preuve. On va analyser l'algorithme pendant une phase. Soit $a = a_0$ la position du fichier adverse au début de la phase et soit a_i sa position après la requête numéro i. Soit m_0 la position du fichier de l'Algorithme MTM au début de cette phase. Le coût de l'Algorithme MTM pendant cette phase est de

$$\begin{aligned}\text{coût}_{\text{MTM}} &= \sum_{i=1}^{D} d(m_0, v_i) + D \cdot d(m_0, m) \\ &\leq \sum_{i=1}^{D} d(a_0, v_i) + D \cdot d(m_0, a_0) + D \cdot d(m_0, m)\end{aligned}$$

(par l'inégalité triangulaire). Le coût de l'adversaire est

$$\text{coût}_{\text{OPT}} = D \cdot \sum_{i=1}^{D} d(a_{i-1}, a_i) + \sum_{i=1}^{D} d(a_{i-1}, v_i).$$

En utilisant l'inegalité triangulaire on obtient pour chaque $0 \leq l \leq D$:

$$\begin{aligned}\text{coût}_{\text{OPT}} &= D \cdot \sum_{i=1}^{D} d(a_{i-1}, a_i) + \sum_{i=1}^{D} d(a_{i-1}, v_i) \\ &\geq \sum_{i=1}^{D} [d(a_l, a_{i-1}) + d(a_{i-1}, v_i)] \geq \sum_{i=1}^{D} d(a_l, v_i).\end{aligned}$$

Donc

$$D \cdot d(a_l, m) \leq \sum_{i=1}^{D} [d(a_l, v_i) + d(v_i, m)] \leq 2 \sum_{i=1}^{D} d(a_l, v_i) \leq 2\text{coût}_{\text{OPT}}$$

(la deuxième inégalité est une conséquence du choix du point m).

On va utiliser la fonction potentiel $\Phi = 2D \cdot d(a, m)$, où a est la position du fichier adverse après cette phase (donc $a = a_D$) :

$$\begin{aligned}
\Delta\Phi &= 2D \cdot (d(a, m) - d(a_0, m_0)) \\
&\leq D(2d(a_0, m) + d(a_0, m) - d(m_0, m) - d(a_0, m_0)) \\
&= 2D \cdot d(a, m) + D \cdot d(a_0, m) + \\
&+ \sum_{i=1}^{D} d(a_0, v_i) - [\sum_{i=1}^{D} d(a_0, v_i) + D \cdot d(m_0, m) + D \cdot d(a_0, m_0)] \\
&\leq 7\text{coût}_{\text{OPT}} - \text{coût}_{\text{MTM}}
\end{aligned}$$

(on utilise l'inégalité triangulaire pour passer de la première ligne à la deuxième et on utilise les expressions établies avant pour obtenir l'inégalité finale). Donc l'Algorithme MTM est 7-compétitif. □

Cours V. Coupes max, coupes min, multicoupes et applications : une histoire de coupe interminable...

5.1. Coupe maximum (COUPE MAX) (`Maximum cut problem`)

Formulation : Étant donnée un graphe $G = (V, E)$, partitionner V en deux sous-ensembles $S, V - S$ tel que le nombre d'arêtes avec une extrémité dans S et l'autre dans $V - S$ soit maximum.

On va utiliser un méthode du type "recherche locale" pour obtenir un algorithme d'approximation. Les solutions voisines d'une solution admissible (V_1, V_2) (donc $V_2 = V - V_1$) sont les partitions (V_{1k}, V_{2k}) où
(i) si $v_k \in V_1$, alors $V_{1k} = V_1 - \{v_k\}$ et $V_{2k} = V_2 \cup \{v_k\}$;
(ii) si $v_k \notin V_1$, alors $V_{1k} = V_1 \cup \{v_k\}$ et $V_{2k} = V_2 - \{v_k\}$.
Pour une partition (V_1, V_2) soit $\delta(V_1) = \delta(V_1, V_2) = \delta(V_2)$ l'ensemble des arêtes avec une extrémité dans V_1 et l'autre dans V_2. Une coupe (V_1, V_2) est un *optimum local* si $|\delta(V_1, V_2)| \geq |\delta(V_{1k}, V_{2k})|$ pour toutes les coupes voisines (V_{1k}, V_{2k}).

Théorème 5.1. *Si (V_1, V_2) est un optimum local, alors $|\delta(V_1, V_2)| \geq \frac{1}{2} OPT$, où OPT est la taille d'une coupe maximale de G.*

Preuve. Soit m le nombre d'arêtes du graphe G. Il est suffisant de montrer que $|\delta(V_1, V_2)| \geq \frac{1}{2} m$. Soient m_1 et m_2 les nombres d'arêtes avec les deux extrémités dans V_1 respectivement V_2. Donc $m = m_1 + m_2 + |\delta(V_1, V_2)|$. Pour une sommet $v_i \in V$ on va définir

$$M_{1i} = \{v : v \in V_1, \text{ et } (v, v_i) \in E\},$$

$$M_{2i} = \{v : v \in V_2 : (v, v_i) \in E\}.$$

Comme (V_1, V_2) est un optimum local, pour chaque sommet v_i on a $|\delta(V_{1i}, V_{2i})| \leq |\delta(V_1, V_2)|$. Donc $|M_{1i}| - |M_{2i}| \leq 0$ si $v_i \in V_1$ et $|M_{2i}| - |M_{1i}| \leq 0$ si $v_i \in V_2$. En prenant la somme sur tous les sommets de V_1 et V_2 on déduit que

$$\sum_{v_i \in V_1} (|M_{1i}| - |M_{2i}|) = 2m_1 - |\delta(V_1, V_2)| \leq 0$$

$$\sum_{v_i \in V_2} (|M_{2i}| - |M_{1i}|) = 2m_2 - |\delta(V_1, V_2)| \leq 0.$$

Donc $m_1 + m_2 - |\delta(V_1, V_2)| \leq 0$. Comme $m_1 + m_2 + |\delta(V_1, V_2)| = m$ on obtient que $|\delta(V_1, V_2)| \geq m/2$.
□

Le meilleur algorithme d'approximation pour COUPE MAX est avec un facteur de 0,878.

5.2. Coupe minimum (COUPE MIN) (`Minimum cut problem`)

Formulation : Étant donné un graphe $G = (V, E)$, chaque arête $e = (x, y)$ avec une capacité $u_e > 0$, partitionner V en deux sous-ensembles $S, V - S$ tel que la somme des capacités des arêtes avec une extrémité dans S et l'autre dans $V - S$ soit minimum.

Pour simplifier les notations, soit $\delta(S) = \delta(S, V - S)$ l'ensemble des arêtes avec une extrémité dans S et l'autre dans $V - S$. Soit $u(\delta(S)) = \sum_{e \in \delta(S)} u_e$ la capacité de $\delta(S)$. Si $u \in S, v \in V - S$ ou $v \in S, u \in V - S$ alors on dit que la coupe $\delta(S)$ *sépare* les sommets u et v, où que $\delta(S)$ est une (u, v)-coupe. Soit $\lambda(G; u, v)$ la capacité d'une (u, v)-coupe minimale. Finalement, soit $\lambda(G)$ la capacité d'une coupe min de G, donc

$$\lambda(G) = min\{\lambda(G; u, v) : u, v \in V, u \neq v\}.$$

Lemme 5.1. *Si $p, q, r \in V$ alors $\lambda(G; p, q) \geq min\{\lambda(G; p, r), \lambda(G; r, q)\}$. Plus généralement, si $p_1, p_2, \ldots, p_k$ sont des sommets distincts de G, alors*

$$\lambda(G; v_1, v_k) \geq min\{\lambda(G; v_1, v_2), \lambda(G; v_2, v_3), \ldots, \lambda(G; v_{k-1}, v_k\}.$$

Preuve. Soit $\delta(S)$ un (p, q)-coupe minimale avec $s \in S$. Si $r \in S$ alors $\delta(S)$ est une (r, q)-coupe, donc $u(\delta(S)) \geq \lambda(G; q, r)$. Sinon $\delta(S)$ est une (p, r)-coupe, donc $u(\delta(S)) \geq \lambda(G; p, r)$. Pour la deuxième affirmation, notons que chaque (v_1, v_k)-coupe min doit séparer deux sommets consécutives v_i et v_{i+1}.
□

Le problème COUPE MIN est le seul problème dans ces notes que l'ont peut résoudre en temps polynomial. Pour cela on peut résoudre un problème de flot maximum pour chaque pair u, v de sommets et trouver la coupe minimale qui sépare u et v, et finallement prendre la meilleur des $O(n^2)$ coupes. Ici on va décrire un algorithme élégant et direct (il est aussi plus efficace) pour trouver la coupe min, ainsi q'un joli algorithme randomisé.

Soient u, v deux sommets distincts de G. Le graphe G_{uv} est obtenu en *identifiant* les sommets u, v. Les arêtes de G_{uv} ont la même capacité que dans G. Cette opération peut créer des arêtes multiples, mais pas de boucles. On pourra remplacer les arêtes parallèles par une seule arête d'une capacité égale à la somme des capacités des arêtes.

On observe facilement que

$$\lambda(G) = min\{\lambda(G_{uv}), \lambda(G; u, v)\}.$$

L'idée consiste à trouver deux sommets u, v telles qu'on puisse calculer $\lambda(G; u, v)$ très simplement. En reitérant cette operation, on trouvera $\lambda(G_{uv})$. Pour cela on va numéroter les sommets du graphe G d'une façon speciale. Un *ordre légal* de G est un ordre total $v_1, v_2, \ldots, v_n$ des sommets de G tel qui

$$u(\delta(V_{i-1}) \cap \delta(v_i)) \geq u(\delta(V_{i-1}) \cap \delta(v_j)) \text{ pour } 2 \leq j \leq n,$$

où $V_i := \{v_1, \ldots, v_i\}$. Autrement dit, à l'itération i le sommet v_i est le sommet non numéroté pour lequel la somme des capacités des arêtes qui joignent ce sommet aux sommets déjà numérotés est maximum. On peut trouver un ordre légal en temps $O(n^2)$.

Lemme 5.2. *Si $v_1, \dots, v_n$ est un ordre légal de G alors $\delta(v_n)$ est une (v_n, v_{n-1})-coupe minimale.*

Preuve. Comme $\delta(v_n)$ est une (v_n, v_{n-1})-coupe, on va montrer que $u(\delta(v_n)) \leq \lambda(G; v_n, v_{n-1})$. Pour cela on utilise une induction sur le nombre d'arêtes et de sommets de G, l'affirmation étant triviale si soit $|V| = 2$ soit $|E| = 0$. Dans le preuve, notons δ' la fonction δ du graphe G'.

Dans le premier cas, supposons que (v_n, v_{n-1}) soit une arête e de G. Soit G' le graphe obtenu en supprimant l'arête e du graphe G. On peut facilement noter que $v_1, v_2 \dots, v_{n-1}, v_n$ est un ordre légal de G'. Par l'hypothèse de l'induction

$$u(\delta(v_n)) = u(\delta'(v_n)) + u_e = \lambda(G'; v_n, v_{n-1}) + u_e = \lambda(G; v_n, v_{n-1}).$$

Maintenant, suposons que v_n et v_{n-1} ne soient pas adjacents en G. Le Lemme 5.1 implique qu'il suffit d'établir que

$$u(\delta(v_n)) \leq \lambda(G; v_n, v_{n-2})$$

et

$$u(\delta(v_n)) \leq \lambda(G, v_{n-1}, v_{n-2}).$$

Pour démontrer la deuxième inegalité on va utiliser l'induction pour le graphe $G' = G \setminus v_{n-1}$. On peut observer que $v_1, \dots, v_{n-2}, v_n$ est un ordre légal de G', donc par hypothèse d'induction

$$u(\delta(v_n)) = u(\delta'(v_n)) = \lambda(G'; v_n, v_{n-2}) = \lambda(G; v_n, v_n).$$

Pour démontrer la première inégalité on va utiliser l'induction pour le graphe $G' = G \setminus v_n$. On peut observer que $v_1, \dots, v_{n-2}, v_{n-1}$ est un ordre légal de G', donc par hypothèse d'induction

$$u(\delta(v_n)) \leq u(\delta(v_{n-1})) = u(\delta'(v_{n-1})) = \lambda(G'; v_{n-1}, v_{n-2}) = \lambda(G; v_{n-1}, v_{n-2})$$

(la première inégalité est due au choix du sommet v_{n-1} dans l'ordre légal). □

Algorithme COUPE MIN

```
M := ∞ et A est nondéfini
Tant que e graphe G contient au moins deux sommets
        Trouver un ordre légal v_1, v_2, ..., v_n de G;
        Si u(δ(v_n)) < M faire M := u(δ(v_n)) et A := δ(v_n);
        Remplacer G par G_{v_n v_{n-1}};
Retourner A.
```

Du Lemme 5.2 on déduit le résultat suivant.

Théorème 5.1 *L'Algorithme COUPE MIN retrouve la coupe min d'un graphe à n sommets en temps $O(n^3)$.*

Dans ce cours nous présentons un seul algorithme randomisé. Il est également destiner à rechercher une coupe min d'un graphe. Soit $u(E) = \sum_{e \in E} u_e$.

Algorithme COUPE MIN RANDOM

```
Tant que G contient au moins deux sommets
       Choisir de façon aléatoire une arête e = (u, v) avec une probabilité
       de u_e/u(E);
       Remplacer G par G_{uv};
Retourner la coupe unique de G.
```

Lemme 5.3 *Soit $A \subset E$ une coupe min de $G = (V, E)$. L'Algorithme COUPE MIN RANDOM va retourner A avec une probabilité $\geq 2/n(n-1)$.*

Preuve. L'algorithme va retourner A si aucune arête de A n'a été contractée. Supposons que i arêtes soient déjà contractées et aucune de ces arêtes n'appartient à A. Soit $G = (V', E')$ le graphe après ces i contractions, donc $|V'| = n - i$. Comme A est une coupe min de G, sa capacité est inférieure ou égale à la moyenne des capacites $u(\delta'(v))$ des coupes $\delta'(v)$ pour $v \in V'$, alors $u(A) \leq 2u(E')/(n-i)$. Donc la probabilité qu'une arête de A soit contractée à l'itération $i+1$ est

$$\frac{u(A)}{u(E')} \leq \frac{2u(E')}{(n-i)u(E')} = \frac{2}{n-i}.$$

Alors la probabilité qu'aucune arête de A ne soit contracté à l'itération $i+1$ est au moins

$$1 - \frac{2}{n-i} = \frac{n-i-2}{n-i},$$

donc la probabilité qu'aucune arête de A ne soit jamais contractée est au moins

$$\frac{n-2}{n} \cdot \frac{n-3}{n-1} \cdot \frac{n-4}{n-2} \cdots \frac{3}{5} \cdot \frac{2}{4} \cdot \frac{1}{3} = \frac{2}{n(n-1)}. \qquad \square$$

Théorème 5.2. *Soit A une coupe min de G et k un entier positif. La probabilité que l'Algorithme COUPE MIN RANDOM ne retourne pas la coupe A après kn^2 exécutions est au maximum e^{-2k}.*

Preuve. Le Lemme 5.2 implique que la probabilité voulue est au maximum

$$(1 - \frac{2}{n(n-1)})^{kn^2} \leq (1 - \frac{2}{n^2})^{kn^2} \leq (e^{-\frac{2}{n^2}})^{kn^2} = e^{-2k}.$$

(on a utilisé l'inégalité $1 - x \leq e^{-x}$ et le fait que la probabilité que l'algorithme ne trouve pas la coupe min A après k exécutions est q^k où q est la probabilité que COUPE MIN RANDOM ne retourne pas la coupe A. $\square$

Par exemple, si on fait tourner l'Algorithme COUPE MIN RANDOM $10n^2$ fois, on trouvera la coupe A avec une probabilité $\geq 0,99999999$. Donc il faut exécuter COUPE MIN RANDOM plusieurs fois et de retourner la plus petite coupe.

5.3. Arbres de coupes (`Cut-trees`)

Formulation : Étant donné un graph $G = (V, E), u_e > 0$ pour chaque arête e et un ensemble de terminaux $K \subseteq V$, trouver une (r, s)-coupe min pour chaque pair $r, s \in K$.

Un façon de résoudre ce problème est de trouver une (r, s)-coupe min pour chaque pair $r, s \in K$ en utilisant le lien entre les coupes min et les flots max. Un solution très élégante a eté proposée par Gomory et Hu. Elle nécessite la résolution de seulement $|K| - 1$ problèmes de flot max obtenir un arbre qui permet de calculer facilement la coupe min pour n'importe quelle pair des terminaux.

On va présenter une description intuitive de l'algorithme. Au début on choisit une paire de terminaux $r, s \in K$ et on trouve la (r, s)-coupe min (R, S) où $s \in S$ et $r \in R$. Nous représentons cette coupe par une arête $e = (R, S)$ d'un arbre T. La longueur f_e de cette arête e est la capacité de la coupe (R, S). Si chacun des R et S contient un seul terminal, alors l'algorithme s'arrête. Sinon, supposons que R contienne deux noeuds terminaux p, q. Soit G_R le graphe obtenu à partir de G en contractant les sommets de S. On trouve la (p, q)-coupe min de G_R. Cette coupe partage l'ensemble R en deux sous-ensembles Q et P, où $p \in P$ et $q \in Q$. On va ajouter S à une des deux parties en suivant la règle : si la partie P de la coupe contient le sommet correspondant à S, alors S est adjacent à P, sinon S est adjacent à Q. Donc l'arbre T courant posséde trois sommets P, Q, S et deux arêtes ((P, Q) et (Q, S) ou (P, Q) et (P, S)). La longueur d'arête (P, Q) est de $\lambda(G_R; p, q)$. En sautant quelques itérations, supposons que T est l'arbre courrant. Les sommets de T sont étiquetés par des sous-ensembles de K

constituant une partition de K. Supposons que un de ces sous-ensembles A contient deux terminaux $y, z \in K$. Soit G_A le graphe obtenu à partir du graphe G en identifiant les sommets de chacune des composantes connexes de la forêt obtenue à partir de T en supprimant le sommet A. Dans G_A on cherche une (y, z)-coupe min, soit (Y, Z). On va former un nouvel arbre en remplacant A par une arête (Y, Z) de longueur $\lambda(G_A; y, z)$. Chacun des voisins du sommet ancien A sera adjacent soit à Y soit à Z, en dépendance de quelle côté dans la coupe est située le sommet identifié respectif. La procédure sera terminée quand tous les sommets de T vont représenter des ensembles avec un seul terminal.

Théorème 5.3. *Pour chaque paire de terminaux $r, s \in K$ la capacité d'une (r, s)-coupe min de G est égale à la longueur f_e de la plus courte arête e du chemin entre s et r de l'arbre T. De plus, la bipartition de V obtenue à partir de T en supprimant l'arête e correspond à une (r, s)-coupe min.*

On commence par quelques résultats auxiliaires. Dans deux premiers résultats par une coupe on comprend une bipartition de V, dans le résultat d'après une coupe est l'ensemble des arêtes avec des extrémités des deux côtés de cette partition.

Lemme 5.4. *Soient $A \subseteq V$ et $B \subseteq V$. Alors*

$$u(\delta(A)) + u(\delta(B)) \geq max\{u(\delta(A \cup B)) + u(\delta(A \cap B)), u(\delta(A - B)) + u(\delta(B - A))\}.$$

Preuve. On va démontrer que $u(\delta(A)) + u(\delta(B)) \geq u(\delta(A \cup B)) + u(\delta(A \cap B))$ (la preuve de la deuxième inegalité est analogue). Pour cela notons que chaque arête des coupes $(A, V - A)$ et $(B, V - B)$ est soit une arête de la coupe $(A \cup B, V - (A \cup B))$ soit une arête de la coupe $(A \cap B, V - (A \cap B))$. □

Deux coupes $(A, V - A)$ et $(B, V - B)$ sont *transversales* si toutes les quatre intersections $A \cap B, A \cap (V - B), (V - A) \cap B, (V - A) \cap (V - B)$ sont non-vides ; sinon on dit qu'elles sont *laminaires*.

Lemme 5.5. *Soient $(A, V - A)$ et $(B, V - A)$ deux coupes min transversales. Alors $(A \cap B, V - (A \cap B))$ et $(A \cup B), V - (A \cup B)$ sont des coupes min laminaires.*

Preuve. Les deux coupes $(A \cap B, V - (A \cap B))$ et $((A \cup B), V - (A \cup B))$ sont laminaires. Le Lemme 5.4 et la minimalité des coupes $(A, V - A)$ et $(B, V - B)$ impliquent que $(A \cap B, V - (A \cap B))$ et $((A \cup B), V - (A \cup B))$ sont aussi des coupes min. □

Lemme 5.6. *Soit $\delta(S)$ une (r, s)-coupe min et $v, w \in S$. Alors il existe une (v, w)-coupe min $\delta(T)$ laminaire avec $\delta(S)$.*

Preuve. Soit $\delta(X)$ une (v, w)-coupe min avec $s \in X$ et suposons que $\delta(S)$ et $\delta(X)$ sont transversales (i.e. $X \cap S \neq \emptyset$, $(V - X) \cap S \neq \emptyset$). Il y a deux cas selon que $r \in X$ où $r \in V - X$. Si $r \in X$, le Lemme 5.4 implique que

$$u(\delta(S \cap (V - X))) + u(\delta(S \cup (V - X))) \leq u(\delta(S)) + u(\delta(V - X)).$$

Puisque $\delta(S \cup (V - X))$ est une (r, s)-coupe, alors

$$u(\delta(S \cup (V - X))) \geq u(\delta(S)).$$

Des deux inégalités on peut déduire que

$$u(\delta(S \cap (V - X))) \leq u(\delta(V - X)) = u(\delta(X)),$$

donc $\delta(S \cap (V - X))$ est une (v, w)-coupe min.

Si $r \in V - X$, le Lemme 5.4 implique que

$$u(\delta(S \cap X)) + u(\delta(S \cup X)) \leq u(\delta(S)) + u(\delta(X)).$$

Comme $\delta(S \cup X)$ est une (r,s)-coupe, donc

$$u(\delta(S \cup X)) \geq u(\delta(S)).$$

Des deux inégalités on peut déduire que

$$u(\delta(S \cap X)) \leq u(\delta(V - X)) = u(\delta(X)),$$

donc $\delta(S \cap X)$ est une (v,w)-coupe min. □

Soit e une arête de l'arbre courrant de Gomory-Hu qui joint les sous-ensembles des terminaux R et S. On dit que deux terminaux $r \in R$ et $s \in S$ sont *représentatifs* pour l'atête e si la longueur f_e de e est égale à $\lambda(G; r, s)$.

Lemme 5.7. *À chaque itération de la procédure de Gomory-Hu chaque arête de T a des paires de terminaux représentatifs.*

Preuve. La propriété est vraie à l'itération numéro 1. On va montrer qu'elle est préservée à chaque itération. Supposons qu'à une itération le noeud A soit partagé en deux sous-ensembles X et Y en utilisant une (x,y)-coupe avec $x \in X$ et $y \in Y$. Le Lemme 5.6 implique que $\lambda(G_A; x, y) = \lambda(G; x, y)$, donc x, y sont des terminaux représentatifs pour l'arête $e = (X, Y)$ de l'arbre T.

Maintenant, soit h une arête de l'arbre précédent qui joint les sous-ensembles A et B et supposons sans pert de généralité que dans T l'arête h joigne B et X. Par hypothèse, il existe deux terminaux $a \in A$ et $b \in B$ représentatifs pour h (donc $l_h = \lambda(G; a, b)$). Si $a \in X$, alors a et b resteront des terminaux représentatifs pour h. Donc $a \in Y$. On va montrer que dans ce cas aussi a et b sont représentatifs pour h. La coupe utilisée pour obtenir l'arête h sépare les sommets x et b, donc $\lambda(G; x, b) \leq \lambda(G; a, b)$. On va montrer l'inégalité inverse.

Soit G' le graphe obtenu à partir de G en identifiant les sommets de Y (soit v_Y ce super-noeud). Le Lemme 5.6 implique que $\lambda(G; x, b) = \lambda(G'; x, b)$. Par ailleurs, $\lambda(G'; x, b) \geq min\{\lambda(G'; x, v_Y), \lambda(G'; v_Y, b)\}$ par le Lemme 5.1. Puisque $a \in Y$, on obtient $\lambda(G'; v_Y, b) \geq \lambda(G; a, b)$. Puisque $y \in Y$ et la (x,y)-coupe qui partage A en X et Y sépare aussi a et b, on déduit que $\lambda(G'; x, v_Y) \geq \lambda(G; x, y) \geq \lambda(G; a, b)$. Les inegalites obtenues impliquent que $\lambda(G; x, b) = \lambda(G'; x, b) \geq \lambda(G; a, b)$ et la démonstration est terminée. □

Preuve du Théorème 5.3. Soit $V_0, e_1, V_1, \dots, e_k, V_k$ un chemin de l'arbre final T qui joint r et s. Il faut montrer que

$$\lambda(G; r, s) = min\{f_{e_1}, f_{e_2}, \dots, f_{e_k}\}.$$

On sait déjà que chacun des ensembles V_i contient un seul terminal, donc $V_i = \{v_i\}$, où $v_0 = r$ et $v_k = s$. L'inégalité $\lambda(G; r, s) \leq min\{f_{e_1}, f_{e_2}, \dots, f_{e_k}\}$ est immédiate puisque chaque coupe correspondant à une arête e_j sépare les sommets r et s. Le Lemme 5.7 implique que $f_{e_i} = \lambda(G; v_{i-1}, v_i)$, $i = 1, \dots, k$. D'autre part, le Lemme 5.1 implique que

$$\lambda(G; r, s) \geq min\{\lambda(G; v_0, v_1), \lambda(G; v_1, v_2), \dots, \lambda(G; v_{k-1}, v_k)\} = min\{f_{e_1}, f_{e_2}, \dots, f_{e_k}\}. \quad \square$$

5.4. Coupes multiterminales (`Multiway cuts`)

Formulation : Étant donné un ensemble de terminaux $S = \{s_1, s_2, \dots, s_k\} \subset V$ d'un graphe $G = (V, E)$, trouver la coupe de capacité minimale qu sépare les terminaux les uns des autres.

Algorithme MULTICOUPE

1. Pour chaque terminal s_i, $i = 1, \dots, k$ trouver la coupe min C_i qui sépare s_i du reste des terminaux ;

2. Écarter la coupe la plus ''lourde'' et retourner l'union C des coupes restantes.

Pour exécuter le premier pas, on peut identifier les terminaux de $S - \{s_i\}$ dans un seule noeud et résoudre le problème de coupe min pour ce noeud et s_i.

Théorème 5.4. *L'Algorithme MULTICOUPE est un algorithme d'approximation avec un facteur de $2 - \frac{1}{k}$ pour le problème de coupe multiterminaux.*

Preuve. Soit A une multicoupe optimale de G. On peut voir A comme une union de k coupes : en supprimant de G les arêtes de A on obtiendra k composantes connexes, chacune avec un terminal (puisque A est une multicoupe optimale, on ne peut pas obtenir plus de k composantes). Soit A_i la coupe qui sépare la composante qui contient s_i du reste du graphe. Alors $A = \cup_{i=1}^{k} A_i$. Chaque arête de A est incidente à deux telles composantes, donc elle sera dans deux coupes A_i. Donc $\sum_{i=1}^{k} u(A_i) = 2u(A)$. Il est clair que A_i est une coupe qui sépare A_i du reste des terminaux, donc $u(C_i) \leq u(A_i)$. Puisque C est obtenue en supprimant la coupe la plus "lourde" C_i, on obtient

$$u(C) \leq (1 - \frac{1}{k}) \sum_{i=1}^{k} u(C_i) \leq (1 - \frac{1}{k}) \sum_{i=1}^{k} u(A_i) = 2(1 - \frac{1}{k}) u(A). \qquad \square$$

Le pire exemple pour l'algorithme MULTICOUPE est un graphe avec $2k$ sommets et $2k$ arêtes constituant un k-circuit plus k terminaux chacun attaché à son sommet du circuit par une arête. Les arêtes du circuit ont la capacité 1, quand aux autres arêtes elles sont toutes de capacité $2 - \epsilon$ pour un ϵ petit. Pour chaque s_i la coupe min sera l'arête incident à s_i, donc la coupe C retournée par l'algorithme est de capacité $(k-1)(2-\epsilon)$. D'autre part, la multicoupe optimale est formée par les arêtes du circuit, donc elle a la capacité k.

5.5. Le problème de k-coupe min (The minimum k-cut problem)

Formulation : Un ensemble d'arêtes qui, si on les supprime, séparent le graphe en k composantes connexes s'appelle une *k-coupe*. Le problème de la k-coupe min est de trouver la k-coupe de capacité minimale.

Algorithme k-COUPE MIN

1. Trouver l'arbre des coupes T de G ;
2. Retourner l'union des $(k-1)$ coupes min les plus légères entre $(n-1)$ coupes associé aux arêtes de T.

Théorème 5.5. *L'Algorithme k-COUPE MIN est un algorithme d'approximation avec un facteur de $2 - \frac{1}{k}$ pour le problème de la k-coupe min.*

Preuve. Soit A la k-coupe optimale de G. Comme dans la preuve du Théorème 5.4, on peut voir A comme une union de k coupes. Soit $V_1, \ldots, V_2, \ldots, V_k$ les k composantes connexes obtenues en supprimant A et soit A_i la coupe qui sépare V_i du reste du graphe. Alors $A = A_1 \cup \ldots \cup A_k$ et puisque chaque arête de A appartient à deux de ces coupes, $\sum_{i=1}^{k} u(A_i) = 2u(A)$. Supposons (sans pert de généralité) que A_k est la coupe la plus lourde. L'idée est d'associer aux coupes $A_1, \ldots, A_{k-1}$ un ensemble de $k-1$ coupes de T et d'établir une bijection entre ces deux ensembles de façon que chaque coupe A_i soit plus lourde ou de même capacité que son partenaire.

Les $k-1$ coupes de T sont définies de la façon suivante. Soit B l'ensemble des arêtes avec des extrémités dans des ensembles différents $V_1, V_2, \ldots, V_k$. Soit $H = (V, B)$ le graphe avec V comme ensemble des sommets et B comme arêtes. On va contracter en H chaque V_i dans un seul sommet. Le graphe obtenu doit être connexe (puisque l'arbre T est connexe). Supprimer les arêtes de ce graphe jusqu'à ce qu'on arrive à un arbre T'. Notons par $B' \subset B$ les arêtes restantes, donc $|B'| = k - 1$.

Les arêtes de B' vont définir les $(k-1)$ coupes voulues. Pour montrer cela, on va enraciner T' en V_k (rappelons que A_k est la coupe la plus lourde entre les A_i). En orientant les arêtes de B' vers la racine, on obtient une correspondance entre les arêtes de B' et les ensembles $V_1, V_2, \ldots, V_{k-1}$.

Supposons que l'arête $(x,y) \in B'$ corresponde à l'ensemble V_i. Comme A_i est une (x,y)-coupe de G, $u(A_i) \geq \lambda(G;x,y)$. Donc chaque coupe de $A_1, \ldots, A_{k-1}$ est au moins aussi lourde que la coupe de G définit par l'arête correspondante de B'. Puisque C est obtenu en prenant les $(k-1)$ coupes les plus légéres de T, alors

$$u(C) \leq \sum_{e \in B'} u(e) \leq \sum_{i=1}^{k-1} u(A_i) \leq (1-\frac{1}{k}) \sum_{i=1}^{k} u(A_i) = 2(1-\frac{1}{k}) u(A). \qquad \square$$

Le pire exemple pour k-COUPE MIN est fourni par le même graphe à $2k$ sommets.

5.6. Arbres phylogénetiques en norme l_∞.

> `Chacun des murs de chaque hexagone porte cinq étagères ; chaque étagère comprend trente-deux livres, tous de même format ; chaque livre a quatre cent dix pages ; chaque page, quarante lignes, et chaque ligne, environ quatre-vingts caractères noirs.`
>
> *La Bibliothèque de Babel* J.L. Borges

Un arbre $T = (V, E, l)$ est défini par l'ensemble des sommets V, par l'ensembles des arêtes E et par la fonction l qui donne la longueur des arêtes. Si $(u,v) \in E$, alors $l(u,v)$ s'appelle la longueur de l'arête (u,v). La longueur d'un chemin de T est la longueur des arêtes qui le composent. On définit la *distance additive* $d_T(x,y)$ la longueur du chemin entre s et t. On dit qu'une distance d sur un ensemble fini S est une *distance arborée* si il existe un arbre $T = (V, E, l)$ tel que $S \subseteq V$ et $d(x,y) = d_T(x,y)$ pour chaque paire $x, y \in S$ (i.e. dans un arbre on prend un sous-ensemble de sommets et les distances additives entre eux comme mesure de distance).

Quelles sont les distances arborées ? En examinant quatre sommets quelconques s,t,u,v d'un arbre, on s'aperçoit que parmis les *trois sommes* $d(s,t)+d(u,v), d(s,u)+d(t,v), d(s,v)+d(t,u)$ *les deux plus grandes sont égales.* Cette condition s'appele la *condition des quatre points.* La preuve du résultat suivant est donnée dans l'Annexe 2.

Théorème 5.6. *Un distance d sur S est une distance arborée si et seulement si elle vérifie la condition des quatre points :*

$$\textit{pour tous } x,y,z \in X, d(x,y)+d(z,t) \leq max\{d(x,z)+d(y,t), d(x,t)+d(y,z)\}.$$

Une sous-classe des distances arborées est formée par les ultramétriques et les distances à centre. Une *distance à centre* est une distance sur S telle que pour chaque sommet $s \in S$ il existe un nombre non-négatif $\delta(s)$ tel que $d(s,t) = \delta(s) + \delta(t)$ pour n'importe quels $s, t \in S$. De façon triviale, une distance à centre est realisable sur une etoile centrée en un sommet c tel que $d(s,c) = \delta(s)$ pour chaque $s \in S$. Une distance arborée d_T sur S est *sphérique* si il existe un point r de T (ou bien r est un sommet de T ou bien un point intérieur d'une arête de T) tel que $d_T(s,r) = d_T(t,r)$ pour tous $s,t \in S$. Une *ultramétrique* est une distance d sur S telle que $d(s,t) \leq max\{d(s,z), d(z,t)\}$ pour tous $s,t,z \in S$.

Proposition 5.1 *Une distance d est une ultramétrique si et seulement si d est une distance arborée sphérique.*

Comme conséquence du ce résultat, les ultramétriques sont répresentées par des hiérarchies indicées (dendrogrammes). On note aussi un lien étroit entre les négalités ultramétriques et l'inégalité du Lemme 5.1 vérifiée par les coupes min (rappelons que les coupes min vérifient l'inégalité anti-ultramétrique $\lambda(G;s,t) \geq min\{\lambda(G;s,z), \lambda(G;z,t)\}$).

Un des problèmes importants de taxonomie numérique, de phylogénetique (mais aussi de linguistique historique et de la biologie d'évolution est le problème d'approximation des distances calculées à partir de données expérimentales (disons la distance génétique entre les ADN où les ARN, la distance entre les textes, etc) par des distances ultramétriques où par des distances arborées. Par exemple, dans le domaine de la biologie d'évolution, en comparant les séquences d'ADN de deux espèces, le biologiste peut estimer le temps passé après la séparation de ces espèces (en supposent que le nombre de mutations est proportionnel à ce temps). Donc une matrice de distances entre chaque deux espèces est construite et à partir de cette matrice et on veut reconstruire l'arbre de l'evolution pour cet ensemble d'espèces. Pour formuler ce problème de façon plus formelle, on doit introduire une mesure de qualité entre les distances données et les distances additives dans l'arbre construit. On voit bien aussi, qui l'hypotèse que la distance arborée obtenue soit une ultramétrique reflète la supposition (souvent critiquée) que la vitèsse d'évolution est constante dans toutes les directions. Dans ce chapitre comme critère d'optimalité on prendra l'erreur uniforme (en norme l_∞) entre une distance d et une distance arborée d_T :

$$\delta(d, d_T) = max_{s,t\in S}|d(s,t) - d_t(s,t)|.$$

Notons par $\mathcal{D}(S)$, $\mathcal{A}(S)$ et $\mathcal{U}(S)$, respectivement, la familles de toutes les distances, la famille de toutes les distances arborées et la famille de toutes les ultramétriques définies sur un ensemble fini S.

Formulation (APPROX PAR DIST ARBORÉES). Étant donnée une distance $d \in \mathcal{D}(S)$ trouver une distance arborée $a \in \mathcal{A}(S)$ que minimise l'erreur uniforme $\delta(d,a)$. Étant donnée une distance $d \in \mathcal{D}(S)$ trouver une distance arborée $a \in \mathcal{A}(S)$ que minimise l'erreur uniforme $\delta(d,a)$.

Formulation (APPROX PAR ULTRAMÉTRIQUES) Étant donnée une distance $d \in \mathcal{D}(S)$ trouver une ultramétrique $u \in \mathcal{U}(S)$ que minimise l'erreur uniforme $\delta(d,u)$.
On va présenter un algorithme polynomial pour résoudre le deuxième problème, ensuite on va s'inspirer de cet algorithme pour obtenir un algorithme d'approximation avec un facteur 3 pour le prmier problème..

Algorithme APPROX ULTRAMÉTRIQUES

1. Trouver l'arbre couvrant T de poids minimum du graphe complet (K_n, d).
2. Pour chaque paire de sommets $s,t \in S$ soit $u_0(s,t)$ la longueur maximale d'une arête sur le chemin entre s et t (par convention, $u_0(s,s) = 0$).
3. Calculer $\delta = \delta(d, u_0) = min_{s,t\in S}|d(s,t) - u_0(s,t)|$.
4. Retourner $u := u_0 + \frac{1}{2}\delta$ (i.e. $u(s,t) = u_0(s,t) + \frac{1}{2}\delta$ pour $s,t \in S$).

Théorème 5.7. *La distance u retournée par l'Algorithme APPROX ULTRAMÉTRIQUES est une ultramétrique optimale pour le problème APPROX PAR ULTRAMÉTRIQUES.*

Preuve. On peut voir facilement que u_0 est une ultramétrique (voir aussi la preuve du Lemme 5.1). Puisque T est un arbre couvrant de poids minimum, $d(s,t)$ est plus petite ou égal à la longueur de n'importe quelle arête du chemin entre s et t dans T, donc $u_0(s,t) \leq d(s,t)$ et $u_0 \leq d$.

Maintenant on va démontrer que $u' \in \mathcal{U}(S)$ et $u' \leq d$ impliquent que $u' \leq u_0$. Considérons une paire s,t d'éléments de S et soit $s = x_0, x_1, \ldots, x_n = t$ le chemin de T entre s et t. On a alors par construction $u_0(s,t) = max\{d(x_i, x_{i+1}) : i = 0, \ldots, n-1\}$. Puisque u', est une ultramétrique, on a $u'(s,t) \leq max\{u'(x_i, x_{i+1}) : i = 1, \ldots, n-1\}$ (voir aussi le Lemme 5.1) et comme $u' \leq d$ on obtient $u'(s,t) \leq max\{d(x_i, x_{i+1}) : i = 1, \ldots, n-1\} = u_0(s,t)$. Donc u_0 est la plus grande des ultramétriques dominées par d; souvent on appele u_0 l'*ultramétrique sous-dominante* de d.

Comme conséquence de cela, u est aussi une ultramétrique. Il reste à montrer qui u est une ultramétrique optimale. Soit $u' \in \mathcal{U}(S)$ une ultramétrique quelconque et soit $\epsilon = \delta(d, u')$. Dans ce cas l'ultramétrique $u'' = u' - \epsilon$ est dominné par d (i.e. $u'' \leq d$), donc $u'' \leq u_0 \leq d$ par la propriété antérieure. De là on déduit que $d(s,t) - u_0(s,t) \leq d(s,t) - u''(s,t) \leq 2\epsilon$, donc $\delta(d, u_0) \leq 2\epsilon$. Comme $\delta(d,u) = \frac{1}{2}\delta(d, u_0)$, on obtient que $\delta(d,u) \leq \epsilon = \delta(d,u')$, donc u est une ultramétrique optimale. □

Soit $d \in \mathcal{D}(S)$ et p un point de S. On dit qu'une distance arborée $a^p \in \mathcal{A}(S)$ est *p-restreinte* si $d(s,p) = a^p(s,p)$ pour n'importe quel point $s \in S$. Si, de plus, a^p minimise l'erreur $\delta(d, a^p)$ dans l'ensemble $\mathcal{A}_p$ de toutes les distances arborées p-restreintes, on dit que a^p est *p-optimale*.

Théorème 5.8. *Soit p un point arbitraire de S et a^p une distance arborée p-optimale. Alors a^p est une 3-approximation pour le problème APPROX PAR DIST ARBORÉE.*

Preuve. Soit $a = d_T$ une solution optimale pour le problème APPROX PAR DIST ARBORÉE. Notons $\gamma(s,t) = d(s,t) - a(s,t)$, donc $\delta := \delta(d,a) = max_{s,t\in S}\{|\gamma(s,t)|\}$. À partir de a on va fabriquer une distance arborée p-restreinte de la façon suivante. Si $\gamma(s,p) < 0$ il faut éloigner le point s de p. Pour cela on va augmenter simplement la longueur de l'arête incidente à s de $|\gamma(s,p)|$ unités. D'autre part, si $\gamma(s,p) > 0$ alors il faut approcher s de p de $\gamma(s,p)$ unités (si l'arête incidente à s est trop courte, le sommet s peut passer quelques sommets internes jusqu'à ce qu'il arrive au bon endroit). En tous cas, le point s se déplace de $|\gamma(s,p)|$ unités. Soit T_0 l'arbre resultant et soit d_{T_0} sa distance additive. La construction implique que d_{T_0} est p-restreint. Pour chaque paire de points $s,t \in S$ on a

$$|d(s,t) - d_{T_0}(s,t)| \le |a(s,t) - d_{T_0}(s,t)| + |d(s,t) - a(s,t)| \le (|\gamma(s,p)| + |\gamma(p,t)|) + |\gamma(s,t)| \le 3\delta(d,a).$$

Puisque a^p est p-optimale, $\delta(d,a^p) \le \delta(d, d_{T_0}) \le 3\delta(d,a)$. $\square$

Le résultat qui suit (donné sans démonstration) montre que les distances arborées p-restreintes ont une certaine ressemblance avec les ultramétriques.

Théorème 5.9. *Soit $d \in \mathcal{D}(S)$ et $p \in S$. Alors d admet une sous-dominante dans l'ensemble des distances arborées p-restreints, donc il existe une distance arborée p-restreintes a_0^p telle que pour n'importe quelle distance arborée p-restreinte $a^p \le d$ on a $a^p \le a_0^p \le d$. La distance $a^p = a_0^p + \frac{1}{2}\delta(d, a_0^p)$ retournée par l'Algorithme APPROX ARBRE p-RESTREINT est une distance arborée p-optimale.*

Algorithme APPROX ARBRE P-RESTREINT

1. `Calculer` d^p `ou`

$$d^p(s,t) = d(s,t) - d(s,p) - d(t,p) \text{ si } s \neq t \text{ et } d^p(s,t) = 0 \text{ si } s = t \qquad (*)$$

2. `Calculer la sous-dominante ultramétrique` u^p `de` d^p `en utilisant l'arbre couvrant de poids minimum.`
3. `Calculer` a_0^p `en utilisant la transformation inverse à (*), i.e.`

$$a_0^p(s,t) = d(s,p) + d(t,p) - u^p(s,t) \text{ si } s \neq t \text{ et } a^p(s,t) = 0 \text{ si } s = t.$$

4. `Calculer` $\delta = \delta(d, a_0^p) = max_{s,t\in S}\{|d(s,t) - a_0^p(s,t)\}|$.
5. `Retourner` $a^p = a_0^p + \frac{1}{2}\delta$ `(i.e.` $a^p(s,t) = a_0^p(s,t) + \frac{1}{2}\delta$`).`

Exercice. Prouver que $d \in \mathcal{A}$ si et seulement si la restriction de d^p sur $S - \{p\}$ vérifie l'inégalité ultramétrique.

Cours VI. Programmation Linéaire pour l'Approximation

La programmation linéaire est la seule méthode efficace qui résout n'importe quel problème en temps exponentiel

6.1. Dualité de la programmation linéaire

Un programme linéaire est un problème qui consiste à optimiser (maximiser ou minimiser) une fonction linéaire sous contraintes linéaires. Par exemple,

$$\begin{array}{lllll} \text{minimiser} & z = 7x_1 + x_2 + 5x_3 & & & \\ & x_1 - x_2 + 3x_3 & \geq & 10 & \\ & 5x_1 + 2x_2 - x_3 & \geq & 6 & (P) \\ & x_1, x_2, x_3 \geq 0. & & & \end{array}$$

est un programme linéaire. La fonction linéaire que l'on doit optimiser est appelée *fonction objectif*. On dit que les nombres $x_1, \ldots, x_n$ qui satisfont toutes les contraintes de (P) constituent une *solution réalisable* de (P). Une solution réalisable qui optimise la fonction objectif est appelée *solution optimale* du problème ; la valeur correspondante de la fonction objectif est appelée la *valeur optimale* du problème. Un grand nombre d'algorithmes d'approximation utilisent la notion de dualité de la programmation linéaire.

Pour illustrer la notion de dualité, au lieu de résoudre (P), nous allons essayer d'obtenir une *estimation* de la valeur optimale z^* de sa fonction objectif. Pour obtenir une *bonne* borne supérieure de z^*, il suffit d'avoir une *bonne* solution réalisable. Par exemple, la solution réalisable $(2, 1, 3)$ donne $z^* \leq 30 = 7.2 + 1 + 5.3$. Par ailleurs, on peut obtenir une borne *inférieure* en comparant terme à terme les coefficients de la première contrainte avec les coefficients de la fonction objectif et en se rappelant que les x_i sont tous positifs, on obtient que pour chaque solution réalisable x, on a

$$7x_1 + x_2 + 5x_3 \geq x_1 - x_3 + 3x_3 \geq 10.$$

Donc la fonction objectif est supérieure à 10 pour toutes les solutions réalisables. Une meilleure borne est obtenue en considérant la somme des deux contraintes,

$$7x_1 + x_2 + 5x_3 \geq (x_1 - x_2 + 3x_3) + (5x_1 + 2x_2 - x_3) \geq 16.$$

Ou encore mieux, en considérant la somme de la seconde et du double de la première,

$$7x_1 + x_2 + 5x_3 \geq 2.(x_1 - x_2 + 3x_3) + (5x_1 + 2x_2 - x_3) \geq 26.$$

D'une façon plus générale, on essaie de construire des *combinaisons linéaires* de contraintes en multipliant la première contrainte par y_1, la seconde par y_2 et en sommant le tout. (Dans le premier cas, on avait $y_1 = 1$, $y_2 = 0$; dans le second $y_1 = 2$, $y_2 = 1$ et finalement $y_1 = 2$, $y_2 = 1$.) L'inégalité correspondante s'écrit

$$y_1(x_1 - x_2 + 3x_3) + y_2(5x_1 + 2x_2 - x_3) \geq 10y_1 + 6y_2$$

ou encore

$$(y_1 + 5y_2)x_1 + (-y_1 + 2y_2)x_2 + (3y_1 - y_3)x_3 \geq 10y_1 + 6y_2. \tag{1}$$

Les y_i doivent être non-négatifs sinon les inégalités changeraient de sens. Par ailleurs, on veut utiliser le membre droit de (1) en tant que borne supérieure de $z = 7x_1 + x_2 + 5x_3$ donc le coefficient de chaque x_j doit être inférieure au coefficient correspondant de z. Plus explicitement, on doit avoir

$$\begin{array}{rcrcl} y_1 & + & 5y_2 & \leq & 7 \\ -y_1 & + & 2y_2 & \leq & 1 \\ 3y_1 & - & y_2 & \leq & 5 \end{array}$$

Si les multiplicateurs y_i sont tous positifs et satisfont ces quatres inégalités, alors on peut conclure que chaque solution (x_1, x_2, x_3) satisfait l'inégalité

$$7x_1 + x_2 + 5x_3 \geq 10y_1 + 6y_2.$$

En particulier, cette inégalité est satisfaite par la solution optimale ; par conséquent

$$z^* \geq 10y_1 + 6y_2.$$

Pour trouver la meilleure borne possible, il faut résoudre le *programme linéaire dual*

$$\begin{array}{lll} \text{maximiser} & 10y_1 + 6y_2 & \\ & y_1 + 5y_2 & \leq \ 7 \\ & -y_1 + 2y_2 & \leq \ 1 \qquad\qquad (D) \\ & 3y_1 - y_2 & \leq \ 5 \\ & y_1, y_2, x_3 \geq 0. & \end{array}$$

Si (y_1, y_2) est une solution de (D) alors $z^* \geq 10y_1 + 6y_2$. Finalement, si on trouve une solution réalisable de xde (P) et une solution réalisable yde (D) telles que

$$10y_1 + 6y_2 = 7x_1 + x_2 + 5x_3$$

alors x est optimal car on sait que

$$10y_1 + 6y_2 \leq z^* \leq 7x_1 + x_2 + 5x_3.$$

Ici x= $(7/4, 0, 11/4)$ et y= $(2, 1)$ forment une paire optimale. En fait, le théorème de la dualité de la programmation linéaire nous dit qu'il est toujours possible de trouver une telle paire de solutions.

Pour formuler précisément ce théorème, considérons un PL sous forme *standard* :

$$\begin{array}{llll} \text{minimiser} & \sum_{j=1}^{n} c_j x_j & & \\ & \sum_{j=1}^{n} a_{ij} x_j \ \geq \ b_i & & i = 1, \dots, m \\ & x_j \geq 0 & & j = 1, \dots, n \end{array}$$

et son dual

$$\begin{array}{llll} \text{maximiser} & \sum_{i=1}^{m} b_i y_i & & \\ & \sum_{i=1}^{m} a_{ij} y_i \ \leq \ c_i & & j = 1, \dots, n \\ & y_i \geq 0 & & i = 1, \dots, m. \end{array}$$

Théorème 6.1. (Dualité de PL) *Le programme primal a un optimum fini si et seulement si le programme dual a un optimum fini. De plus, si $x^* = (x_1^*, \dots, x_n^*)$ et $y^* = (y_1^*, \dots, y_m^*)$ sont solutions optimales respectivement du primal et du dual, alors*

$$\sum_{j=1}^{n} c_j x_j^* = \sum_{i=1}^{m} b_i y_i^*.$$

Seulement une partie de ce théorème nous intéresse dans le cadre des algorithmes d'approximation :

la dualité faible.

Théorème 6.2. (Dualité Faible) *Si $x = (x_1, \ldots, x_n)$ est une solution du réalisable du primal et que $y = (y_1, \ldots, y_m)$ est une solution du réalisable du dual alors*

$$\sum_{j=1}^{n} c_j x_j \leq \sum_{i=1}^{m} b_i y_i.$$

Preuve. Puisque y est une solution réalisable du dual et que les x_j sont tous positifs,

$$\sum_{j=1}^{n} c_j x_j \geq \sum_{j=1}^{n} \left(\sum_{i=1}^{m} a_{ij} y_i \right) x_j. \tag{2}$$

De la même façon, puisque x est une solution réalisable du dual et que les y_i sont positifs

$$\sum_{i=1}^{m} \left(\sum_{j=1}^{n} a_{ij} x_j \right) y_i \geq \sum_{i=1}^{m} b_i y_i. \tag{3}$$

Le théorème est alors démontré en observant que

$$\sum_{i=1}^{m} \left(\sum_{j=1}^{n} a_{ij} x_j \right) y_i = \sum_{j=1}^{n} \left(\sum_{i=1}^{m} a_{ij} y_i \right) x_j \tag{4}$$

□

D'après le théorème de la dualité, pour que xet ysoient optimales il faut et il suffit les inégalités (2) et (3) soient toutes les deux satisfaites à l'égalité. Ce qui conduit au théorème suivant.

Théorème 6.3. (Conditions des écarts complémentaires)
Soit x et y des solutions respectivement du primal et du dual. Alors x et y sont simultanément optimales si et seulement si les conditions suivantes sont satisfaites :

Conditions primales des écarts complémentaires
Pour $1 \leq j \leq n$: ou bien $x_j = 0$, ou bien $\sum_{i=1}^{m} a_{ij} y_i = c_j$;
et
Conditions duales des écarts complémentaires
Pour $1 \leq i \leq m$: ou bien $y_i = 0$, ou bien $\sum_{j=1}^{n} a_{ij} x_i = b_i$.

Avant de présenter des applications de la notion de dualité dans le contexte des algorithmes d'approximation, nous allons illustrer sa portée sur un problème que l'on résout de façon exacte en temps polynômial.

Flot maximum et coupe minimum

Formulation : (Problème du Flot Maximum.) *étant donné un graphe orienté $G = (V, E)$, deux sommets particuliers s la source et t le puis, et des capacités pour les arcs $c : E \rightarrow \mathbb{R}^+$, trouver la quantité maximum de flot qui peut circuler entre s et t de telle sorte que :*

1. *pour chaque arc e, le flot sur l'arc e est inférieur à sa capacité, et*
2. *pour chaque sommet $v \neq s,t$, le flot qui entre en v est égal au flot qui sort de v.*

Une *st-coupe* est définie par une partition des sommets en deux sous-ensembles X et $\bar{X}$ avec $s \in X$ et $t \in \bar{X}$, elle est constitué de tous les arcs dont l'origine est dans X et l'extrémité est $\bar{X}$. La capacité de cette coupe est notée $c(X, \bar{X})$, elle est égale à la somme des capacités de ses arcs.

Le flot entre s et t doit traverser toutes les st-coupes. Par conséquent, la valeur de n'importe quel flot entre s et t est inférieure à la capacité de n'importe quelle st-coupe (dualité faible). Par conséquent, si on trouve un flot et une coupe, tels que la valeur du flot est égale à la capacité de la coupe alors il s'agit d'un flot maximum et d'une coupe de capacité minimum. En fait, on peut toujours trouver une telle paire (dualité forte).

Exprimons le problème du flot maximum comme un programme linéaire. On ajoute un arc fictif entre t et s (capacité ∞) pour se ramener à une *circulation* (conservation du flot pour tous les sommets). f_{ij} représente le flot sur l'arc (i,j). On veut maximiser f_{ts}.

$$
\begin{array}{lll}
\text{maximiser} & f_{ts} & \\
 & f_{ij} \leq c_{ij} & (i,j) \in E \\
 & \displaystyle\sum_{j:(j,i)\in E} f_{ji} - \sum_{j:(i,j)\in E} f_{ij} \leq 0 & i \in V \\
 & f_{ij} \geq 0 & (i,j) \in E
\end{array}
$$

Pour avoir un programme linéaire sous forme standard, on a exprimé la loi de conservation du flot avec une inégalité au lieu d'une égalité. Cela ne change rien car si il existait une solution de ce programme linéaire avec sommet i^+ qui produit du flot (i.e. $\sum_j f_{ji^+} - \sum_j f_{i^+j} < 0$) alors, dans cette solution il y aurait au moins un sommet i^- qui consomme du flot (i.e. $\sum_j f_{ji^-} - \sum_j f_{i^-j} > 0$) et cela violerait une inégalité du programme linéaire.

Pour obtenir le programme linéaire dual, on introduit d_{ij} et p_i les variables duales associées respectivement aux contraintes de capacités et de conservation de flot. On interprète d_{ij} comme une *distance* entre i et j et p_i comme un *potentiel*. Le programme dual s'écrit

$$
\begin{array}{lll}
\text{minimiser} & \displaystyle\sum_{(i,j)\in E} c_{ij} d_{ij} & \\
 & d_{ij} - p_i + p_j \geq 0 & (i,j) \in E \\
 & p_s - p_t \geq 1 & \\
 & d_{ij} \geq 0 & (i,j) \in E \\
 & p_i \geq 0 & i \in V.
\end{array}
$$

Pour interpréter de façon intuitive le dual, il vaut mieux d'abord le transformer en programme linéaire en nombres entiers en remplaçant $d_{ij} \geq 0$ par $d_{ij} \in \{0,1\}$ et $p_i \geq 0$ par $p_i \in \{0,1\}$.

Soit (d^*, p^*) une solution optimale de ce programme linéaire en nombres entiers. Avec une solution en $0/1$, $p_s - p_t \geq 1$ implique $p_s^* = 1$ et $p_t^* = 0$. Par conséquent, à (d^*, p^*) on peut associer naturellement une st-coupe $(X, \bar{X})$, avec $X = \{i \in V \ : \ p_i = 1\}$ et $\bar{X} = \{i \in V \ : \ p_i = 0\}$. Considérons un arc (i,j) avec $i \in X$ et $j \in \bar{X}$. Puisque $p_i^* = 1$ et $p_j^* = 0$, à cause de la première contrainte, $d_{ij}^* \geq 1$. Avec une solution $0/1$, $d_{ij}^* = 1$. La distance des autres arcs peut être affectée à 0 ou 1 sans violer aucune contrainte, mais pour minimiser la fonction objectif elle doit être fixer à 0. Par conséquent la fonction objectif est précisément la capacité de la coupe $(X, \bar{X})$, et $(X, \bar{X})$ doit être une coupe de capacité minimum.

Considérons une *st*-coupe C. L'ensemble C a la propriété que n'importe quel chemin entre s et t contient au moins un élément de C. En utilisant cette propriété, on peut interpreter les solutions réalisables du programme linéaire dual comme des coupes *fractionnaire* : ces solutions sont telles que la somme des distances des arcs de n'importe quel chemin entre s et t doit être supérieure où égale à 1. Pour voir cela, considérons n'importe quel *st*-chemin $(s = v_0, v_1, \ldots, v_k = t)$ et sommons les différences de potentiels entre les extrémités des arcs du chemins

$$\sum_{i=0}^{k-1}(p_i - p_j) = p_s - p_t.$$

Par conséquent, à cause de la première contrainte, la somme des distances des arcs du chemin doit être supérieure à $p_s - p_t$, qui est ≥ 1. On définit la capacité d'une *st*-coupe fractionnaire comme sa valeur de la fonction objectif.

En principe, la meilleure *st*-coupe fractionnaire devrait avoir une capacité plus petite que la meilleure *st*-coupe (entière). En réalité, ce n'est jamais le cas.

Considérons le polyèdre définit comme l'ensemble des solutions réalisables du programme linéaire dual. C'est un résultat classique de programmation linéaire que, pour toutes les fonctions objectifs (affectations de capacités aux arcs), il existe une solution optimale qui est un point extrême de ce polyèdre. De plus, on peut montrer que toutes les sommets de ce polyèdre sont à coordonnées $0/1$. Par conséquent le programme linéaire dual à toujours une solution optimale entière.

Grâce à la dualité forte, la valeur du flot maximum entre s et t doit être égal à la capacité de minimale d'une *st*-coupe fractionnaire. Mais cette dernière est égal à la capacité minimum d'une *st*-coupe, on obtient le théorème du flot max et de la coupe min (*max-flow min-cut theorem*). Par conséquent, ce théorème est un cas particulier de la dualité de la programmation linéaire. En fait, la plupart des relations min-max en optimisation combinatoire sont vérifiées pour des raison analogues.

Finalement, illustrons l'importance des conditions des écarts complémentaires pour obtenir certaines propriétés des flots optimaux et des coupes optimales. Soit f^* une solution optimale du primal (i.e. un flot optimal). Soit (d^*, p^*) une solution optimale entière du dual, et $(X, \bar{X})$ la soupe définit par (d^*, p^*). Considérons un arc (i, j) tel que $i \in X$ et $j \in \bar{X}$. Nous avons déjà prouver que $d^*_{ij} = 1$. Puisque $d^*_{ij} \neq 0$, les conditions des écarts complémentaires imposent que $f^*_{ij} = c_{ij}$. Ensuite, on considère un arc (k, l) tel que $k \in \bar{X}$ et $l \in X$. Puisque $p^*_k - p^*_l = -1$, et $d^*_{kl} \in \{0, 1\}$, la contrainte $d^*_{kl} - p^*_k + p^*_l \geq 0$ doit être satisfaite comme une inégalité stricte. Par conséquent, les conditions primales des écarts complémentaires imposent que $f^*_{kl} = 0$. Donc, nous avons montré que les arcs allant de X vers $\bar{X}$ sont saturés et que les arcs allant de $\bar{X}$ à X ne portent aucun flot.

6.2. Le schéma primal-dual

Considérons un PL sous forme *standard* :

$$\begin{array}{lll} \text{minimiser} & \displaystyle\sum_{j=1}^{n} c_j x_j & \\ & \displaystyle\sum_{j=1}^{n} a_{ij} x_j \quad \geq \quad b_i & i = 1, \ldots, m \\ & x_j \geq 0 & j = 1, \ldots, n \end{array}$$

et son dual

$$\begin{array}{lll} \text{maximiser} & \sum_{i=1}^{m} b_i y_i & \\ & \sum_{i=1}^{m} a_{ij} y_i \quad \leq \quad c_i & j = 1, \ldots, n \\ & y_i \geq 0 & i = 1, \ldots, m. \end{array}$$

Les algorithmes d'approximation de type primal-dual utilisent les conditions primales exactes et ils relaxent les conditions duales des écarts complémentaires :

Conditions primales des écarts complémentaires Pour $1 \leq j \leq n$: ou bien $x_j = 0$, ou bien $\sum_{i=1}^{m} a_{ij} y_i = c_j$;

et

Conditions duales des écarts complémentaires relaxé

Pour $1 \leq i \leq m$: ou bien $y_i = 0$, ou bien $\sum_{j=1}^{n} a_{ij} x_i \leq \alpha \cdot b_i$, où $\alpha > 1$ est une constante.

Lemme 6.1. *Si x et y sont des solutions admissibles primale et duale qui vérifient les conditions des écarts complémentaires ci-dessus, alors*

$$\sum_{j=1}^{n} c_j x_j \leq \alpha \cdot \sum_{i=1}^{m} b_i y_i.$$

Preuve. On a

$$\sum_{j=1}^{n} c_j x_j = \sum_{j=1}^{n} (\sum_{i=1}^{m} a_{ij} y_i) x_j = \sum_{i=1}^{m} (\sum_{j=1}^{n} a_{ij} x_j) y_i \leq \alpha \sum_{i=1}^{m} b_i y_i.$$

La première égalité est obtenue à partir des conditions primales et l'inégalité est une conséquence des conditions duales relaxés. La deuxième égalité est obtenu en échangeant l'ordre de sommation. □

L'algorithme commence avec une solution primale non-admissible et une solution duale admissible ; d'habitude on prend les solutions triviales $x = 0$ et $y = 0$. Ensuite, l'algorithme améliore itérativement l'admissibilité de la solution primale et l'optimalité de la solution duale, en assurent qui à la fin on arrive à une solution primale admissible. La solution primale est entièr à chaque itération, donc la solution finale sera entièr aussi. Les améliorations des solutions primales et duales sont simultanée : la solution courante primale est utilisée pour déterminer l'amélioration de solution duale et vice versa. Le coût de la solution duale est une borne inférieure pour l'OPT, donc à cause de Lemme 6.1 le facteur d'approximation de cet algorithme est α.

Comme illustration, on va présenter un algorithme avec un facteur f pour la Couverture d'Ensemble via le schéma primal-dual. Ce problème et son dual s'écrivent de la façon suivante comme programmes linéares :

$$\text{minimiser} \sum_{S \in \mathcal{S}} \text{coût}(S) x_S$$

$$\sum_{S: e \in S} x_S \geq 1, \qquad e \in U$$

$$x_S \in \{0, 1\}, S \in \mathcal{S},$$

et

$$\text{maximiser} \sum_{e \in U} y_e$$

$$\sum_{e:e\in S} y_e \leq \text{coût}(S), S \in \mathcal{S}$$

$$y_e \geq 0, e \in U.$$

Les conditions primales des écarts complémentaires sont

$$\forall S \in \mathcal{S}: \quad x_S \neq 0 \Rightarrow \sum_{e:e\in S} y_e = \text{coût}(S).$$

L'ensemble S est appellé *serré* si $\sum_{e:e\in S} y_e = c(S)$. Pour améliorer la solution primale on utilisera la condition suivante : *Choisir dans la couverture seulement des ensembles serrés.*

Les conditions duales des écarts complémentaires rélaxées avec $\alpha = f$:

$$\forall e \in U: \quad y_e \neq 0 \Rightarrow \sum_{S:e\in S} x_S \leq f.$$

Puisque x est une solution 0-1, ces conditions sont équivalentes à *Couvrir chaque élément avec un dual non-zéro au maximum f fois.*

Algorithme COUVERTURE d'ENSEMBLE-facteur f

```
1. initialisation : x ← 0; y ← 0;
2. jusqu'à ce que tous les éléments soient couverts faire
     choisir un élément non-couvert e et augmenter y_e jusqu'à ce qu'un ensemble
     devienne serré ;
     choisir tous les ensembles serrés dans la couverture ;
     déclarer tous les éléments dans les ensembles choisis comme ‘‘couverts’’ ;
Retourner les ensembles sélectionnés.
```

Théorème 6.4. *L'Algorithme COUVERTURE d'ENSEMBLE-facteur f est un algorithme d'approximation avec un facteur f.*

Preuve. À la fin de l'algorithme il n'y aura pas d'élément non-couvert ni d'ensemble surchargé, donc les solutions duales et primales sont admissibles. Comme elles vérifient les conditions rélaxées des écarts complémentaires avec $\alpha = f$. Le Lemme 6.1 implique que le facteur d'approximation est f. □

6.2. Coverture Pondéré par des Sommets via PL

Formulation : Etant donné un graphe $G = (V, E)$ et une fonction de poids $w(v)$ sur les sommets de G, trouver une couverture par des sommets $U \subseteq V$ dont le poids $w(U) = \sum_{v\in U} \text{poids}(v)$ est minimal.

Le problème peut être exprimé comme programme linéaire en nombres entiers. Soit $x(v) = 1$ si $v \in U$ et $x(v) = 0$ sinon. Alors

$$OPT = min_{x\in S} \sum_{v\in V} w(v)x(v)$$

où

$$S = \{x \in \mathbb{R}^{|V|} : x(v) + x(u) \geq 1 \quad \forall (u, v) \in E \text{ et } x(v) \in \{0, 1\} \quad \forall v \in V.$$

On va considérer la relaxation de ce programme en nombres entièrs :

$$LB = min_{x\in R} \sum_{v\in V} w(v)x(v)$$

$$R = \{x \in \mathbb{R}^{|V|} : x(v) + x(u) \geq 1 \quad \forall (u,v) \in E \text{ et } x(v) \geq 0 \quad \forall v \in V.$$

Évidemment $S \subseteq R$ donc $LB \leq OPT$. Soit x^* la solution optimale de la relaxation LP. Soit

$$U = \{v \in V : x^*(v) \geq \frac{1}{2}\}.$$

Nous affirmons que U est une solution 2-approximative pour le problème de Couverture Pondéré par des Sommets. L'ensemble U est une couverture par des sommets puisque pour chaque arête (u, v) on a $x^*(u) + x^*(v) \geq 1$, donc ou bien $x^*(u) \geq 1/2$ ou bien $x^*(v) \geq 1/2$. Alors

$$\sum_{v \in U} w(v) \leq \sum_{v \in V} w(v) 2x^*(v) = 2LB$$

puisque $2x^*(v) \geq 1$ pour tous les $v \in U$.

Le schéma primal-dual pour ce problème est analogue à celui de Couverture d'Ensemble. D'abord on va formuler le problème dual : soit $y \in \mathbb{R}^{|E|}$; donc une variable $y(e)$ par arête e du graphe :

$$max \sum_{e \in E} y(e)$$

$$\sum_{u:e=(u,v) \in E} y(e) \leq w(v) \qquad \forall v \in V,$$

$$y(e) \geq 0 \qquad \forall e \in E.$$

Au début soit $C = \emptyset$, $y = 0$ et $F = E$. L'algorithme va répéter les deux pas suivants jusqu'à $F \neq \emptyset$:

1. Choisir $e = (u, v) \in E$. Augmenter $y(e)$ jusqu'à ce que l'inégalité $\sum_{u:e=(u,v) \in E} y(e) \leq w(v) \quad \forall v \in V$ devienne une égalité soit pour u soit pour v, supposons pour u.
2. Ajouter u à C et supprimer de F toutes les arêtes incidents avec u.

C est une couverture par des sommets. De plus,

$$\sum_{v \in C} w(u) = \sum_{v \in C} \sum_{u:e=(v,u) \in E} y(e) = \sum_{e=(v,u) \in E} |C \cap \{v, u\}| y(e) \leq \sum_{e \in E} 2y(e) \leq 2LB.$$

6.3. Multicoupes et multiflots entiers dans les arbres

Formulation : Soit $G = (V, E)$ un graphe avec des capacités $c_e \geq 0 \, \forall e \in E$. Soit $\{(s_1, t_1), \ldots, (s_k, t_k)\}$ une famille des paires de sommets spécifiés. Une *multicoupe* est un ensembles d'arêtes dont la suppression sépare les sommets de chaque paire $\{s_i, t_i\}$. Le problème consiste à trouver la multicoupe de capacité minimum.

Exercice. Montrer qui le problème de Multicoupe min est NP-difficile déjà dans le cas quand $G = (V, E)$ est une étoile avec toutes les arêtes de capacité 1 (conseil : réduire le problème de couverture par des sommets à ce problème).

Donc même dans le cas des arbres le problème est NP-difficile. Pour continuer, soit G un arbre $T = (V, E)$. On introduira une variable d_e pour chaque arête de T qui sera 1 si e est pris dans la multicoupe. Soit p_i le seul chemin de T qui relie les terminaux s_i et t_i. La PL-formulation du problème MULTICOUPE MIN est

$$\text{minimiser} \sum_{e \in E} c_e d_e$$

$$\sum_{e \in p_i} d_e \geq 1, \qquad i \in \{1, \ldots, k\}$$

$$d_e \in \{0,1\}, \qquad e \in E.$$

La relaxation linéaire est obtenue en remplacant la contrainte $d_e \in \{0,1\}$ par $d_e \geq 0$:

$$\text{minimiser} \sum_{e \in E} c_e d_e$$

$$\sum_{e \in p_i} d_e \geq 1, \qquad i \in \{1, \ldots, k\}$$

$$d_e \geq 0, \qquad e \in E.$$

Maintenant, on peut interpréter d_e comme la fraction avec laquelle l'arête e est prise dans la multicoupe. Une solution de ce programme linéaire s'appelle une *multicoupe fractionnaire*. En général, le coût d'une multicoupe fractionnaire peut être plus petit qui celui d'une multicoupe min.

Le programme dual peut être interpréter comme un problème de multiflot, avec une demande par paire (s_i, t_i). La variable duale f_i représente la quantité de flot numéro i sur le chemin entre s_i et t_i.

$$\text{maximiser} \sum_{i=1}^{n} f_i$$

$$\sum_{i: e \in p_i} f_i \leq c_e, \qquad e \in E$$

$$f_i \geq 0, \qquad i \in \{1, \ldots, k\}.$$

Les demandes sont satisfaites de façon concurrente. L'objectif est de maximiser la somme de flots quand le flot total qui circule sur l'arête e est $\leq c_e$. La Dualité Faible (Théorème 6.2) implique qu'un multiflot admissible est une borne inférieure pour la multicoupe min fractionnaire, et par conséquent pour la multicoupe min.

Formulation (MULTIFLOT ENTIER MAX) Soit $G = (V, E)$ un graphe avec des capacités $c_e \geq 0$ $e \in E$. Soit $\{(s_1, t_1), \ldots, (s_k, t_k)\}$ une famille de paires de sommets spécifiés. Une demande est définie pour chaque paire (s_i, t_i). L'objectif est de maximiser la somme des flots entiers avec des constraints de capacité sur chaque arête.

On présentera un algorithme primal-dual qui trouve simultanément une multicoupe et un multiflot entier d'un arbre T avec un facteur de 2 et $\frac{1}{2}$ respectivement. Une arête est *saturée* par un multiflot si la valeur du multiflot qui circule sur cette arête est égale à sa capacité. Les conditions des écarts complémentaires rélaxées sont :

Conditions primales : Pour chaque $e \in E$, si $d_e \neq 0$, alors $\sum_{e \in p_i} f_i = c_e$. De façon équivalente *chaque arête choisie dans la multicoupe doit être saturée.*

Conditions duales rélaxées : pour chaque $i \in \{1, \ldots, k\}$ si $f_i \neq 0$ alors $\sum_{e \in p_i} d_e \leq 2$. De façon équivalente, *choisir au maximum deux arêtes dans un chemin qui porte un flot non-nul.*

Supposons qui l'arbre T est enraciné dans un sommet arbitraire. La *profondeur* d'un sommet v est la longueur du chemin entre v et la racine. Pour deux sommets u, v le plus bas ancêtre commun (lowest common ancestor) $lca(u, v)$ est le sommet de profondeur minimum sur le chemin entre u et v. Soit e_1 et e_2 deux arêtes dans un chemin entre un sommet et la racine. Si e_1 est plus proche de v que e_2 on dira qui e_1 est *plus profonde* que e_2.

L'algorithme commence par une multicoupe et un multiflot vides. Dans une itération, il choisit le sommet non-traité le plus profond, disons v, et il va définir de façon gloutonne des flots entre les paires s_i, t_i telles qui $v = lca(s_i, t_i)$. Quand il n'est plus possible de definir un flot entre deux sommets, les arêtes devenues saturées sont ajouter à la liste D. Après quel'on ait considéré tous les sommets, D sera une multicoupe ; mais il y aura peut-être des arêtes redontantes. Pour les supprimer, on fait un parcours inverse de la liste D : les arêtes sont considerées dans l'ordre inverse de leur insertion dans D et si, après que l'on ait supprimé une arête e, D reste une multicoupe, alors cette arête est écartée de D.

Algorithme MULTICOUPE-MULTIFLOTS DANS LES ARBRES

1. initialisation : $f \leftarrow 0, D \leftarrow 0$
2. pour chaque sommet v parcuru dans l'ordre croissant de profondeur, faire
   pour chaque paire (s_i, t_i) telle que $lca(s_i, t_i) = v$ définir de façon gloutonne un
   flot intégral entre s_i et t_i;
   ajouter à D toutes les arêtes qui devienentt saturées, dans un ordre arbitraire;
3. soit $e_1, e_2, \ldots, e_l$ la liste ordonnée des arêtes de D;
4. suppression en parcours inverse : pour $j = l$ à 1 faire
   si $D - \{e_j\}$ est une multicoupe de T alors, $D \leftarrow D - \{e_j\}$;
5. Retourner le multiflot f et la multicoupe D.

Lemme 6.2. *Soit (s_i, t_i) une paire avec un flot non-nul et soit $lca(s_i, t_i) = v$. Alors chaque chemin entre v et les terminaux s_i, t_i peut contenir au maximum une arête choisie dans la multicoupe.*

Preuve. Supposons que le chemin entre v et s_i contienne deux arêtes e et e' de D et que e soit plus profonde que e'. On va considérer le moment du parcours inverse où l'arête e est testée. Puisque e n'est pas supprimé, il doit exister une paire, disons (s_j, t_j) telle que e est la seule arête de D sur le chemin entre s_j et t_j. Soit $u = lca(s_j, t_j)$. L'arête e' n'appartient pas au (s_j, t_j)-chemin, donc u doit être plus profonde que e', donc plus profonde que v. Après que u a été analysé, D doit contenir une arête sur le chemin entre s_j et t_j, disons l'arête e''. Comme le flot entre s_i et t_i est non-nul, e a été ajouté à D pendant ou après que le sommet v soit analysé. Comme v est un ancêtre de u, l'arête e est ajoutée après l'arête e''. Donc e'' appartient à D au moment où l'arête e est testé. Ça contredit la supposition que à ce moment e est la seule arête de D du chemin entre s_j et t_j. □

Théorème 6.5. *L'Algorithme MULTICOUPE-MULTIFLOT DANS LES ARBRES est un algorithme d'approximation avec un facteur 2 pour le problème de Multicoupe et avec un facteur $\frac{1}{2}$ pour le problème de MULTIFLOT ENTIER dans les arbres. En particulier, pour les arbres avec des capacites entières*

$$max_{flow\ entier\ F} F \leq min_{multicoupe\ C} |C| \leq 2 \cdot max_{flow\ entier\ F} F.$$

Preuve. Le flot défini à la fin du pas 2 est maximal et parce que à ce moment D contient toutes les arêtes saturées, alors D est une multicoupe. Pendant le parcours inverse on supprime seulement des arêtes redondantes, D reste une multicoupe, donc D et f sont des solutions admissibles pour chacun des problèmes.

Comme chaque arête de la multicoupe est saturé, les conditions primales sont vérifiées. D'après le Lemme 6.2 chaque chemin avec un flot non-nul contient au maximum deux arêtes de D. Donc les conditions duales rélaxées sont aussi verifiées. Le Lemme 6.1 implique que la capacité de la multicoupe est maximum deux fois plus petite que le flot. Comme un flot admissible est une borne inférieure pour une multicoupe optimale, et une multicoupe admissible est une borne supérieure pour un multiflot entier optimal, le résultat est démontré. □

Pour les graphes généraux, dans le dernier cours on présentera un algorithme pour le problème de multicoupe avec un facteur $O(logk)$. D'autre part, il n'existe aucun algorithme d'approximation pour le problème du multiflot entier.

Cours VII. Multicoupe dans les graphes généraux

6.1. Formulation

Etant donné un graphe $G = (V, E)$ et k paires de sommets $(s_1, t_1), \ldots, (s_k, t_k)$, une *multicoupe* est un sous-ensemble de d'arêtes $F \subseteq E$, dont la suppression déconnecte toutes les paires de sommets (s_i, t_i) pour $i = 1, \ldots, k$. Dans le problème de la multicoupe minimum, un capacité positive c_e est également donnée pour chaque arête $e \in E$, et on veut trouver une multicoupe de capacité minimum (une multicoupe min pour aller vite). Nous allons présenter un algorithme d'approximation avec un facteur $O(\log k)$ pour ce problème.

On a vu que le problème de la coupe min se résout en calculant une succession de flot maximum. Il n'est pas possible d'opérer de la même façon pour le problème de la multicoupe min. Cependant, l'algorithme d'approximation présenté ici commence par résoudre le problème de multiflot maximum dont la formulation suit. Etant donné un graphe $G = (V, E)$ et k paires de sommets $(s_1, t_1), \ldots, (s_k, t_k)$ (les sommets s_i et t_i sont à interpréter comme une source et une destination entre lesquels on veut faire circuler un certain produit i), l'objectif est de maximiser la somme des quantités de produits acheminées. Le produit i circule entre sa source s_i et sa destination t_i en vérifiant une loi de conservation en chaque sommet intermédiaire. La quantité totale, tous produits confondus, qui circulent sur une arête dans les deux directions ne doit pas excéder la capacité de l'arête.

Commençons par donner une formulation de ce problème en programmation linéaire. Pour chaque produit i, on note P_i l'ensemble de tous les chemins dans G entre s_i et t_i, et $P = \bigcup_{i=1}^k P_i$. Ce programme linéaire aura une variable f_p pour chaque $p \in P$, qui représente la quantité de flot qui circule sur le chemin p. L'objectif est de maximiser la somme des flots routés sur les chemins en respectant les capacités des arêtes (ici, les lois de conservations de flots sont prises en compte de façon implicite). Ce programme linéaire a un nombre exponentiel de variables ; cependant ça ne pose pas de problème ici puisqu'il s'agit d'abord d'obtenir une formulation propre du problème dual. De toute façon, nous expliquerons plus bas comment on peut le résoudre malgrès tout en temps polynomial.

$$\begin{array}{lll} \text{maximiser} & \displaystyle\sum_{p \in P} f_p & \\ \text{avec} & \displaystyle\sum_{p : e \in p} f_p \leq c_e, & e \in E \\ & f_p \geq 0, & p \in P. \end{array} \tag{7.1}$$

Pour écrire le dual de ce programme linéaire, on note d_e la variable duale associée à l'arête e ; nous allons interpréter ces variables comme des longueurs d'arêtes.

$$\begin{array}{lll} \text{minimiser} & \displaystyle\sum_{e \in E} c_e d_e & \\ \text{avec} & \displaystyle\sum_{e \in p} d_e \geq 1, & p \in P \\ & d_e \geq 0, & e \in E. \end{array} \tag{7.2}$$

Le programme dual consiste à trouver des longueurs pour les arêtes de telle sorte que la longueur de chaque chemin entre une source et une destination soit au moins 1, i.e. une affectation des longueurs des arêtes est une solution admissible pour le dual si et seulement pour $i = 1, \ldots, k$ le plus court chemin entre s_i et t_i a une longueur au moins égale à 1.

Intuitivement, pour visualiser le problème, on peut associer au graphe un système de tubes, avec un tube pour chaque arête, les sommets étant les jonctions entre les arêtes. La longueur de chaque tube est donnée par d_e et sa section par c_e. Le problème de la multicoupe fractionnaire consiste à trouver des longueurs pour les tubes de telle sorte que chaque paire source destination soient à distance 1 au minimum et que le volume total $\sum_{e_i n E} c_e d_e$ du système de tube soit minimum.

Une solution optimale entière de (7.2) est une multicoupe optimale. Une solution optimale fractionnaire de (7.2) peut être interprétée comme une multicoupe fractionnaire. Par la dualité de la programmation linéaire, la multicoupe min est égal au multiflot max. Cette valeur peut-être strictement plus petite que la multicoupe optimale entière.

Cela amène naturellement la question de savoir si le rapport entre la multicoupe min et le multiflot max est borné. Maintenant nous allons présenter un algorithme pour trouver une multicoupe avec un facteur $O(\log k)$ par rapport au multiflot max, ce qui montre que le gap est borné par $O(\log k)$.

6.2. Algorithme

D'abord notons que le dual (7.2) se résout en temps polynomial en utilisant l'algorithme l'ellipsoïde. Pour cela, il suffit d'avoir un oracle séparateur qui étant donnée une solution ou bien affirme qu'elle satisfait toutes les contraintes du dual ou bien trouve une inégalité qu'elle ne vérifie pas. Dans notre cas, il faut calculer la longueur du plus court chemin entre chaque paire de sommets (s_i, t_i), si elle est toujours supérieure ou égal à 1, la solution est admissible. Si on trouve un chemin dont la longueur est inférieure ou égale à 1, on a une inégalité violée. Notons d_e la longueur de l'arête e dans une multicoupe optimale fractionnaire et $F = \sum_{e \in E} c_e d_e$ la valeur optimale correspondante.

Notre objectif est de trouver un ensemble d'arêtes dont la somme des capacités n'est pas trop grand par rapport à F et qui soit une multicoupe. Soit D l'ensemble des arêtes dont les longueurs sont non nulles, i.e. $D = \{e : d_e > 0\}$. Clairement D est une multicoupe ; cependant, la capacité de D peut être grande par rapport à F. Comment choisir dans D un sous-ensemble d'arêtes de petite capacité qui soit encore une multicoupe ? Puisque la multicoupe fractionnaire est la façon la plus économique de déconnecter toutes les paires source-destination, il est naturel de donner préférence aux arêtes dont la longueur est grande. C'est ce que fait indirectement l'algorithme que nous allons décrire maintenant.

L'algorithme travaille sur le graphe $G = (V, E)$ dont les longueurs des arêtes sont données par d_e. Soit $\mathrm{dist}(u, v)$ la longueur entre u et v dans ce graphe. Pour un ensemble de sommets $S \in V$, on note $\delta(S)$ l'ensemble des arêtes dans la coupe $(S, V - S)$, $c(\delta(S))$ la capacité de cette coupe et finalement on note $\mathrm{wt}(S)$ le poids de S qui est *approximativement* $\sum c_e d_e$, la somme portant sur toutes les arêtes dont au moins une des extrémités est dans S (une définition plus précise du poids sera donnée plus bas).

L'algorithme va trouver plusieurs ensembles de sommets disjoints, $S_1, \ldots, S_l$, dans G, appelées *régions*, tels que :

- Aucune région ne contient une paire source-destination, et pour chaque i, ou bien s_i ou bien t_i appartient à une des régions.
- Pour chaque région S_i, $c(\delta(S_i)) \leq \epsilon \ \mathrm{wt}(S_i)$ où ϵ est un paramètre que l'on définira plus tard.

Grâce à la premièrement conditions, l'union des coupes de ces régions, i.e. $M = \delta(S_1) \cup \delta(S_2) \cup \ldots \delta(S_l)$, est une multicoupe, et grâce à la deuxième condition sa capacité $c(M) \leq 2\epsilon F$. (Le facteur 2 apparait parce qu'une arête peut contribuer aux poids d'au plus deux régions, plus tard nous pourrons nous débarrasser de ce facteur).

Faire croître une région : le processus continu. Nous allons maintenant fixer le paramètre ϵ de telle sorte que les deux conditions ci-dessus puissent être vérifées ensembles ; de plus, on veut le plus petit ϵ qui suffit. Pour cela, il est plus commode de considérer un processus continu de croissance des régions ; l'algorithme utilisera une version discrète de ce processus.

Chaque région est obtenue en faisant croître un ensemble, en commençant avec un seul sommet, la source ou la destination d'une paire, par commodité on commencera toujours avec une source. Celle-ci sera appelée la *racine* de la région. Voyons comment construire la région S dont la racine est s_1. Pour chaque rayon r on définit $S(r)$ comme l'ensemble de tous les sommets à distance au plus r de s_1, i.e., $S(r) = \{v : \mathrm{dist}(s_1, v) \leq r\}$. $S(0) = \{s_1\}$, et alors que r croît de façon continu à partir de 0, en certains points discrets, $S(r)$ croît en ajoutant des sommets dans l'ordre croissant de leur distance jusqu'à s_1.

Lemme 7.1 *Si le processus de croissance se termine avant que le rayon deviennent $\frac{1}{2}$, alors l'ensemble S ne contient pas de paire source destination.*

Preuve. La distance entre toute paire de sommets dans $S(r)$ est $\leq 2r$. Puisque pour chaque paire (s_i, t_i), $\operatorname{dist}(s_i, t_i) \geq 1$ le lemme est établi. □

Pour des raisons techniques qui seront justifiées par la suite, on affecte un poids à la racine, $\operatorname{wt}(s_1) = \frac{F}{k}$. Pour une arête e qui a au moins une de ses extrémités dans $S(r)$, q_e désigne la *fraction* de l'arête e qui est dans $S(r)$. Si les deux extrémités de e sont dans $S(r)$ alors $q_e = 1$. Sinon, supposons $e = (u, v)$ avec $u \in S(r)$ et $v \notin S(r)$. Alors,

$$q_e = \frac{r - \operatorname{dist}(s_1, u)}{\operatorname{dist}(s_1, v) - \operatorname{dist}(s_1, u)}.$$

Pour le processus continu le poids d'une région est défini comme suit

$$\operatorname{wt}(S(r)) = \operatorname{wt}(s_1) + \sum c_e d_e q_e,$$

où la somme porte sur toutes les arêtes qui ont au moins une extrémité dans $S(r)$. Nous modifierons cette définition légèrement pour le processus discret.

Nous voulons garantir que l'on aura $c(S(r)) \leq \epsilon \operatorname{wt}(S(r))$ pour $r < \frac{1}{2}$. Clairement, on peut toujours garantir cela en prenant un ϵ suffisament grand ; nous montrons ci-dessous comment prendre un ϵ petit.

Lemme 7.2 *Fixer $\epsilon = 2\ln(k+1)$ suffit pour assurer que la condition $c(S(r)) \leq \epsilon wt(S(r))$ sera réalisée avant que le rayon atteigne $\frac{1}{2}$.*

Preuve. Raisonnons par l'absurde. Supposons que tout au long du processus de croissance de la région, en commençant avec $r = 0$ et en finissant avec $r = \frac{1}{2}$, $c(S(r)) > \epsilon \operatorname{wt}(S(r))$. A chaque instant, le différentiel du poids d'une région s'écrit

$$d\operatorname{wt}(S(r)) = \sum_e c_e d_e \; dq_e$$

Seules les arêtes qui ont une extrémité dans $S(r)$ contribuent à la somme. Considérons une arête de ce type $e = (u, v)$ telle que $u \in S(r)$ et $v \notin S(r)$. Alors,

$$c_e d_e \; dq_e = c_e \frac{d_e}{\operatorname{dist}(s_1, v) - \operatorname{dist}(s_1, u)} dr.$$

Puisque $\operatorname{dist}(s_1, v) \leq \operatorname{dist}(s_1, u) + d_e$, on obtient $d_e \geq \operatorname{dist}(s_1, v) - \operatorname{dist}(s_1, u)$, et par conséquent $c_e d_e \; dq_e \geq c_e dr$. Cela donne

$$d\operatorname{wt}(S(r)) \geq c(S(r))dr > \epsilon \operatorname{wt}(S(r))dr.$$

Le poids initial de la région est $\frac{F}{k}$ et le poids final est au plus $F + \frac{F}{k}$. En intégrant, on obtient

$$\int_{\frac{F}{k}}^{F+\frac{F}{k}} \frac{1}{\operatorname{wt}(S(r))} \; d\operatorname{wt}(S(r)) > \int_0^{\frac{1}{2}} \epsilon \; dr.$$

Et donc, $\ln(k+1) > \frac{1}{2}\epsilon$. Mais cela contredit l'affirmation que $\epsilon = 2\ln(k+1)$, ce qui termine la preuve. □

Le processus discret. Le processus discret commence avec $S = \{s_1\}$, et ajoute des sommets dans l'ordre croissant de leur distance jusqu'à s_1. Donc, essentiellement, il consiste à exécuter un algorithme de plus court chemin à partir de la racine. Evidemment, les ensembles de sommets trouvés par les deux processus sont les mêmes.

Pour le processus discret, on redéfinit le poids de la région S comme suit :

$$\mathrm{wt}(S) = \mathrm{wt}(s_1) + \sum_e c_e d_e,$$

où la somme porte sur toutes les arêtes qui ont au moins une extrémité dans S, et $\mathrm{wt}(s_1) = \frac{F}{k}$. Le processus discret s'arrête dès que $c(\delta(S)) \leq \epsilon \mathrm{wt}(S)$, où ϵ est encore $2\ln(k+1)$. Notons que pour le même ensemble S, $\mathrm{wt}(S)$ dans le processus discret est au moins aussi grand que dans le processus continu. Par conséquent, le processus discret ne peut pas terminer avec un ensemble plus grand que le processus continu. Donc, l'ensemble S trouvé ne contient aucune paire source-destination.

Construire successivement les régions. La première région est construite dans le graphe G, en commençant avec n'importe quelle source comme racine. Les autres régions sont construites l'une après l'autre. Soit $G_1 = G$ et S_1 la région trouvée dans le graphe G_1. Après un certain nombre d'étapes les régions $S_1, \ldots, S_{i-1}$ ont été construites. Maintenant, on considère le graphe G_i obtenu en supprimant de G les sommets $S_1 \cup \ldots \cup S_{i-1}$, et les arêtes adjacentes à ces sommets.

Si G ne contient aucune paire source-destination, on a fini. Sinon, on choisit la source d'une telle paire, disons s_j, comme racine, on fixe son poids à $\frac{F}{k}$, et on fait croître une région autour dans G_i. Dans ce qui suit toutes les définitions, telles que la distance ou le poids, seront relatives au graphe G_i. Comme précédemment, la valeur de ϵ est $2\ln(k+1)$, et la condition d'arrêt est $c(\delta_{G_i}(S_i)) \leq \epsilon \mathrm{wt}_{G_i}(S_i)$. Notons qu'à chaque itération la racine est le seul sommet qui a un poids non nul.

De cette manière, nous allons trouver des régions $S_1, \ldots, S_l$, $l \leq k$, et renvoyer comme solution l'ensemble $M = \delta_{S_1} \cup \ldots \cup \delta_{S_l}$. Puisque les arêtes de chaque coupe sont supprimées du graphe à chaque étape, les ensembles de cette union sont disjoints, et $c(M) = \sum_i c(\delta_{G_i}(S_i))$.

L'algorithme est résumé ci-dessous. Notons que pendant qu'une région croît, les longues arêtes vont rester dans la coupe plus longtemps, et auront plus de chance d'apparaître dans la coupe. (En réalité, le temps exact pendant lequel une arête reste dans la coupe dépend de la différence entre les distances de ces extrémités jusqu'à la racine.) Par conséquent, comme promis, l'algorithme donne indirectement une certaine préférence aux arêtes longues.

Lemme 7.3 *L'ensemble M obtenu est une multicoupe*

Preuve. On doit montrer qu'aucune région ne contient une paire source-destination. A chaque itération i, la somme des poids des arêtes du graphe et le poids de la racine courrante est bornée par $F + \frac{F}{k}$. Par conséquent, avec la preuve du Lemme 2, on est assuré que le processus de croissance de la région rencontrera sa condition d'arrêt avant que le rayon soit égal à $\frac{1}{2}$. Par conséquent, la distance entre une paire de sommets dans la région S_i trouver par le processus discret est inférieure ou égale à 1. Nous avons défini ces distances dans le graphe G_i. Puisque G_i est un sous-graphe de G, la distance entre deux sommets dans G est inférieure ou égale à la distance entre ces deux sommets dans G_i. Par conséquent, S_i ne contient aucune paire source-destination. □

Algorithme MulticoupeMin

1. `Trouver une solution optimale pour le PL (7.2), en obtenant ainsi des longueurs pour chaque arêtes de` G.
2. $\epsilon \leftarrow 2\ln(k+1)$, $H \leftarrow G$, $M \leftarrow \emptyset$;
3. `Tant qu'il existe une paire source-destination dans` H `faire :`
   `Choisir une telle source, disons` s_j ;
   `Faire croître une région` S `dont la racine est` s_j `tant que` $c(\delta_H(S)) \leq \epsilon$ $\mathtt{wt}_H(S)$;
   $M \leftarrow M \cup \delta_H(S)$;
   $H \leftarrow H$ `avec les sommets` S `en moins` ;
4. `Retourner` M ;

Lemme 7.4 $c(M) \leq 2\epsilon F = 4\ln(k+1)F$.

Preuve. A chaque itération i, à cause des conditions d'arrêts, on a $c(\delta_{G_i}(S_i)) \leq \epsilon \mathrm{wt}_{G_i}(S_i)$. Puisque toutes les arêtes qui contribuent à $\mathrm{wt}_{G_i}(S_i)$ sont supprimées du graphe après cette itération, chaque arête de G contribue au poids d'au plus une région. Le poids total des arêtes de G est F. Puisque chaque itération consiste à déconnecter au moins une paire source-destination, le nombre d'itération est bornée par k. Par conséquent, le poids total attribué aux sommets racines est au plus F. En ajoutant tout, on obtient

$$c(M) = \sum_i c(\delta_{G_i}(S_i)) \leq \epsilon \left(\sum_i \mathrm{wt}_{G_i}(S_i) \right) \leq \epsilon \left(k\frac{F}{k} + \sum_e c_e d_e \right) = 2\epsilon F.$$

□

Théorème 7.5 *L'algorithme MulticoupeMin est un algorithme d'approximation avec un facteur $O(\log k)$ pour le problème de la multicoupe minimum.*

Preuve. La preuve vient du Lemmes 7.3 et du Lemme 7.4, et du fait que la valeur d'une multicoupe minimum fractionnaire est une borne inférieure pour la multicoupe minimum. □

Corollaire 7.6 *Dans un graphe non-orienté avec k paires source-destination,*

$$\max_{\text{multiflot } F} |F| \leq \min_{\text{multicoupe } C} \leq O(\log k) \left(\max_{\text{multiflot } F} |F| \right)$$

où $|F|$ représente la valeur du multiflot F, et $|C|$ représente la capacité de la multicoupe C.

Exemple. Nous allons construire une famille infinie de graphes pour lesquels le gap d'intégralité du programme linéaire (7.2) est $\Omega(\log k)$, en montrant ainsi que notre analyse de l'Algorithme MulticoupeMin et que le théorème max-multiflot min-multicoupe approché présenté dans le Corrolaire 7.6 sont serrés à un facteur constant près.

La construction utilise des graphes *expansés.* Un graphe $G = (V, E)$ est dit expansé si pour chaque ensemble $S \in V$ avec $|S| \leq \frac{|V|}{2}$, $|\delta(S)| \geq |S|$. Pour une constante appropriée d, il existe une infinité de graphes expansés dont les sommets sont tous de degré d. Soit H un tel graphe contenant k sommets.

Les paires source-destination sont choisies dans H de la façon suivante. Considérons un arbre de parcours en largeur enraciné dans un certain sommet v. Le nombre de sommets à distance $\alpha - 1$ du sommet v est au plus $1 + d + d^2 + d^{\alpha-1} < d^\alpha$. Choisir $\lfloor \log_d \frac{k}{2} \rfloor$ assure qu'au moins $\frac{k}{2}$ sommets sont à distance $\geq \alpha$ de v. Une paire de sommets sera une paire source-destination si la distance qui les séparent est au moins α. Par conséquent, on a choisit $\Omega(k^2)$ paires source-destination.

Chaque arête dans H a une capacité unitaire. Donc, la capacité totale des arêtes de H est $O(k)$. Puisque la distance entre chaque paire source-destination est $\Omega(\log k)$, n'importe quel unité de flot qui circule sur un chemin consomme $\Omega(\log k)$ unité de capacité. Par conséquent, la valeur du multiflot maximum dans H est bornée par $O(\frac{k}{\log k})$. Maintenant nous allons montrer qu'une multicoupe minimum dans H a une capacité $\Omega(k)$ prouvant ainsi que le gap d'intégralité $O(\log k)$ est atteint. Considérons les composantes connexes obtenues en supprimant la multicoupe M de H.

Assertion 7.7 *Chaque composante connexe à au moins $\frac{k}{2}$ sommets.*

Preuve. Supposons qu'une composante connexe est plus de $\frac{k}{2}$ sommets. Considérons un sommet quelconque v dans cette composante. A cause de l'argument présenté plus haut, le nombre de sommets qui sont à une distance $\alpha - 1$ de v dans le graphe H entier est $< d^\alpha \leq \frac{k}{2}$. Donc, il existe un sommet u dans la composante à distance au moins α de v, i.e., u et v forment une paire source-destination. Par conséquent, la suppression de M n'a pas séparé toutes les paires source-destination ce qui conduit à une contradiction. □

A cause de l'Assertion 7.7 et du fait que H est expansé, chaque composante S est telle que $|\delta(S)| \geq |S|$. Puisque chaque sommet de H est dans l'une des composantes, $\sum_S |\delta(S)| \geq \frac{k}{2}$, où la somme porte sur toutes les composantes connexes. Puisqu'une arête contribue aux coupes d'au plus deux composantes, on obtient que le nombre d'arêtes dont les extrémités sont dans des composantes différentes est $\Omega(k)$. Cela donne la borne inférieure voulue pour la valeur d'une multicoupe minimum.

Maintenant, vérifions que le nombre de paires source-destination définies dans le graphe ne dépend pas du nombre de sommets du graphe. Notons qu'en remplaçant une arête par un chemin dont les arêtes aurait une capacité unitaire, on ne change ni la valeur du flot maximum ni de la capacité de la multicoupe min. En utilisant cette opération on peut construire, à partir du graphe H, un graphe G ayant n sommets, pour $n \geq k$ quelconque. Le gap d'intégralité pour le PL (7.2) est donc $\Omega(\log k)$.

Cours VIII. Problèmes de connexité dans les graphes.

Un graphe $G = (V, E)$ s'appelle *k-arête-connexe* si en supprimant k arêtes quelconques on obtient un graphe connexe. Un graphe $G = (V, E)$ s'appelle *k-sommet-connexe* si en suppriment k sommets quelconques on obtient un graphe connexe. Un *pont* est une arête dont la suppression déconnecte le graphe G.

8.1. Sous-graphes k-connexes d'un graphe

Formulation. Soit $G = (V, E)$ un graphe avec n sommets et m arêtes, trouver un sous-graphe $H = (V, E_H)$ de G k-arête-connexe avec un nombre minimum d'arêtes.

On va présenter un algorithme d'approximation dans le cas $k = 2$. L'idée de l'algorithme est la suivante. On parcours le graphe G en profondeur (Depth First Search). Comme résultat on obtient un arbre T enraciné avec $n-1$ arêtes. Les sommets de T sont numérotées en ordre préfixe. Les arêtes restantes de G s'appellent des *arêtes de retour* (une extrémité d'une telle arête est l'ancêtre de l'autre extrémité). Les arêtes de T sont pris dans E_H. Pendant la recherche en profondeur certaines arêtes de retour seront prises dans E_H. Une arête de retour peut être choisie juste avant le dernier parcours d'un sommet. Avant de traverser un sommet v pour la dernière fois, on vérifie que l'arête $(v, p(v))$ entre v et son père dans T est un pont ou non. Si $(v, p(v))$ est encore un pont du graphe H courrant, alors on va "couvrir" cette arête en ajoutant à E_H l'arête de retour entre l'ancêtre $l(v)$ de v avec le plus petit numero qui est adjacent dans G avec un descendant de v et ce descendant $s(v)$. Pour implémenter cet algorithme les structures des données seront utilisées :
prefixe(v) : le numéro attribué au sommet v quand il est visité pour la première fois.
etat(v) : au début chaque sommet est *nonvisité*; après la première visite d'un sommet il devient *decouvert*; quand on sort de sommet pour la dernière fois il est *terminé*.
l(v) : l'ancêtre de v avec le plus petit numéro qui est adjacent dans G avec un descendant de v.
s(v) : le descendent de v qui est la deuxième extrémité de l'arête de retour incidente à $l(v)$.

Avant de donner l'analyse de cet algorithme, on va établir un résultat auxilaire. Un *découpage arborescent de taille* d'un graphe $G = (V, E)$ est une partition $V_1, V_2, \ldots, V_k$ de l'ensemble des sommets V qui vérifie les conditions suivantes. Chaque sous-ensemble est un noeud d'un arbre Γ. Pour un sommet quelconque $v \in V_j$ les voisins de v dans G soit appartiennent à V_j soit à un noeud V_i adjacent dans l'arbre Γ avec V_j.

Lemme 8.1. *Si le graphe G admet un découpage arborescent de taille k alors le nombre minimum d'arêtes dans un sous-graphe 2-arête connexe de G est de* $2(k-1)$.

Preuve. L'arbre Γ contient $k-1$ arêtes. Chaque telle arête $e = (V_i, V_j)$ divise l'arbre Γ en deux sous-arbres Γ_1 et Γ_2. Supposons que $V_i \in \Gamma_1$ et $V_j \in \Gamma_2$. Cette bipartition définit une bipartition des sommets de G dans deux sous-ensembles S_e et $V - S_e$. Dans un sous-graphe 2-arête-connexe H on doit avoir au moins deux arêtes entre S_e et $V - S_e$. Par définition d'un découpage arborescent, chaque telle arête aura un extrémité dans V_i et l'autre dans V_j. Comme les sous-ensembles V_i sont deux à deux disjoints, pour chaque arête e de Γ il y a au moins deux arêtes distinctes dans H, donc le nombre total d'arêtes de H est $\geq 2(k-1)$. □

Étant donné l'arbre T de parcours en profondeur du graphe G, on définit une partition spéciale des sommets de G qui s'appelle une partition DFS. Certains appels recursifs aboutiront a ajouter une arête de type $(s(v), l(v))$ à E_H, d'autres non. Dans chaque cas quand une arête de retour a été ajoutée à E_H, supprimer l'arête correspondante de T; les composantes connexes résultantes de T constituront la partition DFS.

Lemme 8.2. *La partition DFS induit un tissu arborescent de G.*

Preuve. Soit (v_1, v_2) une arête de retour de D. Supposons que dans la partition DFS, v_1 appartient à V_i et v_2 appartient à V_j. Supposons que v_1 est un ancêtre de v_2. De façon évidente, $l(v_2) \leq v_1$. L'algorithme implique que l'on supprime au maximum une arête de T sur le chemin entre v_1 et v_2. Alors, ou bien $V_i = V_j$ ou bien V_i est le père de V_j dans Γ. □

Théorème 8.1. *L'algoritme presenté est un algorithme avec un facteur $\frac{3}{2}$.*

Preuve. Dans le graphe H, il y aura $(n-1)$ arêtes de T plus $k-1$ arêtes de retour, où k est la taille du tissu arborescent. Donc, $|E_H| = n-1+k-1$. Soit OPT(G) le nombre d'arêtes dans une solution optimale. Alors OPT(G) $\geq max\{n, 2(k-1)\}$, puisque d'une côté le nombre minimum d'arêtes dans un graphe 2-connexe avec n sommets est $\geq n$ et de l'autre côté le Lemme 1 implique qu'il est $\geq 2(k-1)$. Donc le rapport est de $\leq \frac{n-1+k-1}{max\{n,2(k-1)\}}$. Si $n \geq 2(k-1)$, ce rapport est evidemment $< 3/2$. Si $n \leq 2(k-1)$, on déduit aussi qui le rapport est $< 3/2$. □

8.2. Augmentation de connexité sous contrainte de diamètre

> -Non, dit le petit prince. Je cherche des amis. Qu'est-ce que signifie "apprivoiser" ?
>
> -C'est une chose trop oubliée, dit le renard. Ça signifie "créer des liens..."
>
> *Le Petit Prince* A. de Saint-Exupéry

Le problème d'augmenter un graphe pour satisfaire certaines contraintes de connexité en ajoutant de nouvelles arêtes est un problème important pour la conception de réseaux résistants aux pannes. Dans la version la plus élémentaire, étant donné un graphe non orienté $G = (V, E)$ on doit ajouter un nombre minimum d'arêtes E' de telle sorte que le graphe augmenté $G = (V, E \cup E')$ soit biconnexe. Eswaran et Tarjan ont caractérisé le nombre minimum d'arêtes E' qui doivent être ajoutées, donnant ainsi un algorithme linéaire pour ce problème. Dans cette partie du cours nous considérons les problèmes d'augmentation de connexité avec une contrainte supplémentaire de diamètre (cette version apparait aussi dans le contexte des réseaux de télécommunications).

Problème ACCD (AUGMENTATION de CONNEXITE sous CONTRAINTE de DIAMETRE) : *étant donné un graphe $G = (V, E)$ et un entier positif D, ajouter un nombre minimum d'arêtes E' de telle sorte que le graphe augmenté $G' = (V, E \cup E')$ soit biconnexe et de diamètre au plus D.*

Le problème ACCD est NP-complet pour les forêts et les arbres. De plus, ce problème et le problème voisin

Problème ACD (AUGMENTATION sous CONTRAINTE de DIAMETRE) : *étant donné un graphe G et un entier positif D, ajouter un nombre minimum d'arêtes pour obtenir un graphe de diamètre au plus D*

sont NP-difficiles pour chaque $D \geq 2$ fixé. Ici nous présentons un algorithme d'approximation pour ACCD (et pour ACD) dans les forêts et les arbres. De plus, nous montrons que pour les arbres on peut résoudre en temps polynomial la version "rayon" de ACD :

Problème ACR (AUGMENTATION sous CONTRAINTE de RAYON) : *étant donné un graphe G et un entier positif R, ajouter un nombre minimum d'arêtes tels que le graphe augmenté ait un rayon au plus R.*

On peut trouver une solution optimale de ACR en résolvant le problème suivant pour chaque sommet et en choisissant une solution avec un nombre minimum d'arêtes :

Problème ACE(G, R, b) **(AUGMENTATION sous CONTRAINTE d'ECCENTRICITE) :** *Etant donné un graphe $G = (V, E)$, un entier positif R, et un sommet $b \in V$, trouver une augmentation minimale E' telle que dans le graphe $G' = (V, E \cup E')$ l'eccentricité de b soit au plus R.*

Concluons cette partie introductive avec quelques définitions nécessaires. Soient un graphe $G = (V, E)$ et deux sommets $u, v \in V$, on note $d_G(u, v)$ la *distance* entre ces deux sommets (si u et v sont dans des composantes connexes distinctes de G, on fixe $d_G(u, v) = \infty$). L'*eccentricité* $e_G(u)$ du sommet u est la distance de u au sommet le plus éloigné de u. Le *diamètre* $\mathrm{diam}(G)$ et le *rayon* $\mathrm{rad}(G)$ de G sont respectivement la plus grande et la plus petite eccentricité des sommets de G. Pour un entier positif k et un sommet $u \in V$, on note

$$B(u, k) = \{v \in V : d_G(u, v) \leq k\}$$

la boule de centre u et de rayon k.

Notre solution est basée essentiellement sur un algorithme polynomial qui résout le problème de k-domination sur les arbres et la relation entre ce problème et le problème ACE.

Formulation (kDOMINATION) Etant donné un graphe $G = (V, E)$ et un ensemble $X \subseteq V$, le problème de k-domination consiste à trouver une couverture de X avec un nombre minimum de boules de rayon k.

Ce problème se résout en temps polynomial pour les arbres $T = (V, E)$ de façon suivant. Le graphe *d'intersection des k-boules* de T est le graphe $\Gamma_k(T) = (V, F)$ dans lequel deux sommets u et v sont adjacent si et seulement si les k-boules $B(u, k)$ et $B(v, k)$ s'intersectent.

Exercice 1. Démontrer que le graphe $\Gamma_k(T)$ est triangulé (i.e. chaque circuit de longueur ≥ 4 posséde une corde).

Exercice 2. Démontrer que la famille des toutes les k-boules d'un arbre T verifie la propriété de Helly : toutes les sous-familles de k-boules deux à deux intersectantes ont une intersection non vide.

Si on va considérer une clique C du graphe $\Gamma_k(T)$, la définition de ce graphe implique que $B(u, k) \cap B(v, k) \neq \emptyset$ pour chaque pair des sommets $u, v \in C$. La propriété de Helly implique q'il existe une sommet commun $x \in \cap_{u \in C} B(u, k)$. Comme $d_G(x, u) \leq B$ pour chaque $u \in C$, on déduit qui $C \subseteq B(x, k)$. Donc le problème de k-domination en T est équivalent avec le problème de couverture de V avec un nombre minimum des cliques du graphe $\Gamma_k(T)$. Puisque $\Gamma_k(T)$ est triangulé, le problème de couverture de clique est resolvable en temps polynomial (voir l'Annexe 3).

Revenons à notre problèmes de départ. D'abord, nous avons besoin du lemme suivant qui établit que le problème ACE admet toujours une solution de type "étoile".

Lemme 8.3. *Etant donné un graph H, un sommet b, et un entier positif r, il existe une solution optimale du problème $ACE(H, r, b)$ composée seulement d'arêtes incidentes à b.*

Preuve. Soit E' une solution optimale quelconque de ACE. Notons H' le graphe obtenu à partir de H en ajoutant les arêtes de E'. Toute arête $uv \in E'$, $u, v \neq b$, appartient à un plus court chemin de H' (de longueur au plus r) entre b et un certain sommet q (sinon nous on pourrait diminuer E' en supprimant l'arête uv). Ceci implique que les sommets u et v ne sont pas équidistants de b. Supposons $d_{H'}(u, b) < d_{H'}(v, b)$. Remplaçons dans E' toute arête uv par l'arête vb. Puisque dans le nouveau graphe H'' on a encore $d_{H''}(b, q) \leq r$, nous obtenons une solution optimale de $\mathrm{ACE}(H, r, b)$ sous la forme d'une étoile centrée en b. $\square$

Continuons en montrant comment réduire le problème $\mathrm{ACE}(G, R, b)$ au problème de k-domination. Pour cela, posons $X := V - B(b, R)$ et $k := R - 1$. Pour une augmentation E' de G qui induit une étoile centrée en b notons $S' = \{s : bs \in E'\}$ et $G' = (V, E \cup E')$ (par le lemme 1, une telle solution de type étoile existe toujours).

Lemme 8.4. *L'ensemble des nouvelles arêtes E' est une solution de ACE(G, R, b) si et seulement si S' est une solution d'une certaine instance du problème de k-domination.*

Preuve. Si l'ensemble X est couvert par les boules $B(s, R-1), s \in S'$, alors pour tous les sommets $x \in X$ il existe un sommet $s \in S$ tel que $d_{G'}(x, b) \leq d_G(x, s) + 1 \leq R$. Notons que s n'est pas adjacent à b dans G, sinon $x \in B(b, R)$. Par conséquent E' est une solution admissible du problème d'eccentricité. Réciproquement, si l'eccentricité de b dans le graphe augmenté G' est au plus R et $v \in V$, alors ou bien $d_G(b, v) \leq R$ et alors v appartient à la boule $B(b, R)$, ou bien un plus court chemin de G' entre v et b utilise une nouvelle arête bs. Puisque ce chemin ne contient pas d'autres nouvelles arêtes, on a $d_G(s, v) \leq R - 1$. Cela montre que S' est une solution pour le problème de k-domination. $\square$

Pour résoudre le problème ACR sur un graphe G, on doit résoudre ACE(G, R, b) pour tous les sommets b et choisir parmi toutes les solutions l'augmentation admissible avec le nombre minimum d'arêtes. La complexité de cette procédure est $O(|V||E|)$ pour chaque graphe G dans lequel le problème de k-domination se résout en temps linéaire $O(|E|)$, en particulier pour les arbres.

Lemma 8.5. *Si T est un arbre alors une solution optimale pour le problème ACR construite par l'algorithme précédent est une approximation avec un facteur 2 pour le problème ACD avec $D = 2R$.*

Preuve. Evidemment, si on prend une solution admissible pour ACR sous la forme d'une étoile, alors le graphe augmenté a un diamètre inférieur à $2R$. Réciproquement, soient E' une solution optimale de ACD et G' le graphe augmenté. Notons P l'ensemble des extrémités des arêtes de E'. Plaçons autour de chaque sommet $p \in P$ une boule $B(p, R-1)$, et appelons Q l'ensemble de tous les sommets de G non couverts par de telles boules, i.e., $Q := V - \bigcup\{B(p, R-1) : p \in P\}$. Nous affirmons que le diamètre de Q est au plus D, en d'autres termes, que $d_G(u, v) \leq D$ pour tous $u, v \in Q$. Supposons le contraire, $d_G(u, v) > D$ pour certains sommets $u, v \in Q$. Puisque $d_{G'}(u, v) \leq D$, tous les plus courts chemins dans G' entre u et v utilisent au moins une arête ajoutée. Cela signifie qu'un tel chemin est composé de trois parties : un sous-chemin de u jusqu'au premier sommet p_u de P, suivi d'une partie composée d'un sous-chemin de G et d'une ou plusieurs arêtes ajoutées se terminant avec le dernier sommet p_v de P, et finalement, un sous-chemin de G à partir de p_v jusqu'à v. Puisque $u \notin B(p_u, R-1)$ et $v \notin B(p_v, R-1)$, on déduit que $d_{G'}(u, v) \geq R + 1 + R > D$, ce qui contredit le choix de E'. Par conséquent la distance dans G entre deux sommets quelconques de Q est au plus $2R$, donc les boules $B(q, R), q \in Q$, s'intersectent deux à deux. Puisque G est un graphe de Helly, ces boules ont un sommet b en commun (qui n'est pas nécessairement un sommet de Q). Soit $E'' = \{bp : p \in P\} - E$. Evidemment E'' est une solution admissible du problème ACE(G, R, b). Puique $|E''| \leq 2|E'|$ (le cas pire intervient lorsque E' est un couplage sur P), on obtient l'inégalité voulue. $\square$

Théorème 8.2. *Il existe un algorithme d'approximation avec un facteur 3 pour le problème ACCD avec $D = 2R$ sur les forêts et les arbres avec un complexité $O(|V|^2)$.*

Preuve. L'algorithme construit une solution admissible en résolvant séparément le problème d'augmentation de biconnexité et le problème ACR. Soit c' le nombre d'arêtes utilisées pour biconnecter le graphe et c'' le nombre d'arêtes qui sont utiles pour résoudre le problème d'augmentation sous contraintes de rayon. Supposons qu'une solution optimale pour ACCD ait c arêtes. Puisque $c \geq c'$ et $c'' \leq 2c$ par le Lemme 3, on a le résultat. $\square$

Cours IX. Problèmes de minmax

9.1. Le problème de k-clusterisation

Formulation. Soit $S = \{s_1, s_2, \ldots, s_n\}$ un ensemble de points dans un espace métrique (X, d) et k un entier positif. Partitionner S en k classes (clusters) $S_1, \ldots, S_k$ de façon que le plus grande diamètre d'une classe soit minimum (rappelons que le *diamètre* d'une classe est $diam(S_i) = max\{d(s_j, s_l) : s_j, s_l \in S_i\}$.

Un algorithme très simple est utilisé. On choisit un point arbitraire s_{i_1} dans la premièr classe S_1. Choisir le point s_{i_2} le plus éloigner de s_{i_1} et placer ce point dans S_2. Après choisir le point s_{i_3} qui maximise la distance jusqu'àu le plus proche entre les points s_{i_1} et s_{i_2} et placer s_{i_3} dans S_3. Continuer ce processus k fois, a chaque itération j choisir le point s_{i_j} qui maximise $\min\{d(s_{i_j}, s_{i_1}), d(s_{i_j}, s_{i_2}), \ldots, d(s_{i_j}, s_{i_{j-1}})\}$ et placer s_{i_j} dans la classe S_j. Les points $s_{i_1}, \ldots, s_{i_k}$ s'appellent les *représentants* des classes $S_1, \ldots, S_k$. Une fois que les représentants sont choisi, la partition de S est défini comme suit : le cluster S_j est formé de tous les points plus proches de s_{i_j} que des autres représentants de classes.

Théorème 9.1. *L'algorithme est un algorithme d'approximation avec un facteur 2.*

Preuve. Soit $s_{i_{k+1}}$ le point qui maximise

$$\delta = \min\{d(s_{i_{k+1}}, s_{i_1}), d(s_{i_{k+1}}, s_{i_2}), \ldots, d(s_{i_{k+1}}, s_{i_k})\}.$$

Par le choix des représentants des classes, la distance entre chaque paire de points $s_{i_1}, \ldots, s_{i_k}, s_{i_{k+1}}$ est au moins δ. Dans chaque partition de S en k classes, deux de ces points seront dans une classe commune, donc OPT$\geq \delta$. Soit $s_i, s_l \in S_j$. Evidemment $d(s_{i_j}, s_i) \leq \delta, d(_{i_j}, s_l) \leq \delta$, donc $d(s_i, s_l) \leq 2\delta$. Comme conséquence, $diam(S_j) \leq 2\delta \leq 2\text{OPT}$. □

9.1. Le problème de k-centres

Formulation. Soit $V = \{s_1, s_2, \ldots, s_n\}$ l'ensemble des points d'un espace métrique (V, d) fini et k un entier positif. Partitionner V en k classes (clusters) $V_1, \ldots, V_k$ de façon à ce que le plus grand rayon d'une classe soit minimum (rappelons que le *rayon* d'une classe est $r(V_i) = min_{c \in V} max\{d(c, s_i) : s_i \in V_i\}$. En d'autres termes, trouver un sous-ensemble C de k points de V qui minimise $max_{s_i \in V}\{min_{c \in C}\{d(s_i, c)\}\}$.

On peut considérer (V, d) comme un graphe complet $G = (V, E)$ où la longueur d'une arête $e = (u, v)$ est $d(e) = d(u, v)$. Supposons que les arêtes de G sont indexées dans l'ordre croissant des longueurs : $d(e_1) \leq d(e_2) \leq \cdots \leq d(e_m)$. Notons $G_i = (V, E_i)$, où $E_i = \{e_1, e_2, \ldots, e_i\}$. Rappelons qu'un *sous-ensemble dominant* d'un graphe $H = (V, E_H)$ est un sous-ensemble $D \subseteq V$ tel que chaque sommet de $V - D$ est adjacent avec un sommet de D. Soit $\text{dom}(H)$ la taille du plus petit sous-ensemble dominant de H. Le problème de k-centres est équivalent au problème de trouver le plus petit indice i tel que le graphe G_i contient un sous-ensemble dominant de taille $\leq k$. Si i^* est le plus petit indice, alors $d(e_{i^*})$ est le rayon optimal dans le problème de k-centres ; on va noter cette valeur par OPT.

Le *carré H^2 d'un graphe H* contient une arête (u, v) pour chaque paire de sommets u et v connecté dans H par une arête ou par un chemin de longueur deux.

Lemme 9.1. *Soit I un sous-ensemble stable de H^2. Alors, $|I| \leq dom(H)$.*

Preuve. Soit D le sous-ensemble dominant de H de taille minimum. Donc H contient $|D|$ étoiles qui couvrent tous les sommets de H. Chaque étoile est une clique (sous-graphe complet) dans H^2, donc H^2 contient $|D|$ cliques qui couvrent tous les sommets. Comme I peut contenir au maximum un sommet dans chaque clique, $|I| \leq |D|$. □

Algorithme kCENTRES.

1. `Construire les graphes` $G_1^2, G_2^2, \ldots, G_m^2$.
2. `Calculer un stable maximal (par inclusion)` M_i `dans chaque graphe` G_i^2.
3. `Trouver le plus petit indice` i `tel que` $|M_i| \leq k$, `disons` j.
4. `Retourner` M_j.

Lemme 9.2. *Soit j définit par l'Algorithme kCENTRES. Alors,* $d(e_j) \leq$OPT.

Preuve. Pour chaque $i < j$ on a $|M_i| > k$. Le Lemme 9.1 implique que $dom(G_i) > k$, donc $i^* > i$. Donc $j \leq i^*$. □

Théorème 9.2. *L'Algorithme kCENTRES est un algorithme d'approximation avec un facteur 2.*

Preuve. Notons qu'un stable maximal I est aussi un sous-ensemble dominant : dans le cas contraire si le sommet v n'est pas dominé par I alors $I \cup \{v\}$ est aussi un stable, ce qui contredit le choix de I comme stable maximal. En particulier, M_j est un sous-ensemble dominant de G_j^2, donc les étoiles de G_j^2 centrées en M_j couvrent tous les sommets de V. Par l'inégalité triangulaire, les arêtes utilisées dans ces étoiles sont de longueur $\leq 2 \cdot d(e_j) \leq 2 \cdot$OPT (par le Lemme 9.2). □

Théorème 9.3. *Si P≠NP, alors il n'existe pas d'algorithme polynomial d'approximation pour le problème de k-centres avec un facteur $2 - \epsilon$, $\epsilon > 0$.*

Preuve. On va montrer qu'un tel algorithme va résoudre le problème (NP-complet) de sous-ensemble dominant. Soit $G = (V, E)$ une instance du problème de sous-ensemble dominant et k un entier. Construire un graphe $G' = (V, E')$ comme suit : $d(u, v) = 1$ si (u, v) est une arête de G et $d(u, v) = 2$ sinon. Si $dom(G) \leq k$, alors G' contient un k-centre de rayon 1. D'autre part, si $dom(G) > k$, alors le k-centre optimal est de rayon 2. Dans le premier cas, une algorithme avec un facteur $2 - \epsilon$ d'approximation va trouver un k-centre de rayon 1, puisque il ne peut pas utiliser d'arêtes de longueur 2. Donc en utilisant un tel algorithme on pourra faire la distinction entre les deux possibilités et on peut résoudre le problème de sous-ensemble dominant. □

Donc la question d'un algorithme d'approximation pour le problème de k-centres avec un meilleur facteur est close. On continue avec une généralisation de ce problème.

Formulation (Le problème de centres pondérés). En plus de la fonction de longueur sur les arêtes, soit $w : V \to \mathbb{R}^+$ une fonction de poids sur le sommets et $W \in \mathbb{R}^+$ une borne. Le problème consiste à trouver un sous-ensemble de centres $C \subseteq V$ de poids total $\leq W$ qui minimise la fonction objectif $max_{v \in V}\{min_{c \in C}\{d(c, v)\}\}$.

Soit pdom(G) le poids d'un sous-ensemble dominant de G de poids minimum. Pour les graphes $G_1, G_2, \ldots, G_m$ defini avant, il faut trouver le plus petit indice i tel que pdom$(G_i) \leq W$. Si i^* est cet indice, alors OPT=$d(e_{i^*})$.

Pour un graphe H avec des poids sur les sommets, soit I un stable de H^2. Pour chaque $u \in I$ soit $s(u)$ le voisin de u en H de poids minimum (aussi le sommet u est considéré comme voisin de u). Soit $S = \{s(u) : u \in I\}$.

Lemme 9.3. $\sum_{v \in S} w(v) =: w(S) \leq wdom(H)$.

Preuve. Soit D le sous-ensemble dominant de poids minimum. On peut trouver un ensemble d'étoiles disjointes centrées dans les sommets de D qui couvrent tous les sommets de H. Chaque étoile est une clique (sous-graphe complet) dans H^2, donc H^2 contient $|D|$ cliques qui couvrent tous les sommets. I peut contenir au maximum un sommet dans chaque clique, donc chaque sommet de I contient le centre de l'étoile qui le contient comme voisin dans H. Donc $w(S) \leq w(D)$. □

Algorithme CENTRES PONDERÉS.

1. `Construire les graphes` $G_1^2, G_2^2, \ldots, G_m^2$.
2. `Calculer un stable maximal (par inclusion)` M_i `dans chaque graphe` G_i^2.
3. `Calculer` $S_i = \{s_i(u) : u \in M_i\}$.
4. `Trouver le plus petit indice` i `tel que` $w(S_i) \leq W$, `disons` j.
5. `Retourner` S_j.

Théorème 9.4. *L'Algorithme CENTRES PONDÉRÉS est un algorithme d'approximation avec un facteur 3.*

Preuve. Le Lemme 9.3 donne une borne inférieure pour OPT : $d(e_j) \leq$OPT (voir le Lemme 9.2). Puisque M_j est une sous-ensemble dominant de G_j^2, on peut couvrir V avec des étoiles de G_j^2 centrées dans les sommets de M_j. L'inégalité triangulaire implique que les étoiles utilisent des arêtes de longueur $\leq 2 \cdot d(e_j)$. Chaque centre d'une étoile est adjacent avec un sommet de S_j par une arête de longueur $\leq d(e_j)$. On va déplacer chaque centre d'étoile dans son voisin de S_j et on va redéfinir les étoiles. Une fois de plus, l'inégalité triangulaire implique que la longueur de chaque arête dans les étoiles finales est $\leq 3 \cdot d(e_j)$. □

Références

[1] V. Vazirani, *Approximation Algorithms,* Springer 2000
`http ://www.cc.gatech.edu/fac/Vijaj.Vazirani/`

[2] *Approximation Algorithms for NP-hard Problems,* D. S. Hochbaum (ed.), PWS Publishing Company, 1997.

[3] M. Goemans, *Advanced Algorithms,* course notes
`http ://www-math.mit.edu/ goemans/#notes`

[4] J. Cherian, R. Ravi, *Algorithms for Network Problems*
`http ://www.gsia.cmu.edu/afs/andrew/gsia/ravi/`

[5] A. Borodin, R. El-Yaniv, *Online Computation and Competitive Analysis,* Cambridge University Press, 1998.

[6] *Online Algorithms. The State of the Art,* A. Fiat, G.J. Woeginger (eds) Springer 1998.

[7] W.J.Cook, W.H. Cunningham, W.R. Pulleyblank, A. Schrijver, *Combinatorial Optimization,* Wiley 1998.

[8] G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela, M. Protasi, *Complexity and Approximation. Combinatorial Problems and their Approximability Properties,* Springer 1999.

[9] Th. Cormen, Ch. Leiserson, R. Rivest, *Introduction à l'Algorithmique,* Dunod 1994.

Annexe I : Le dessin esthétique des graphes

Graphs like women like to look nicely

Proverb ouzbek (traduction libre de l'auteur)

...mais je me rappelai alors que j'avais surtout étudié la géographie, l'histoire, le calcul et la grammaire et je dis au petit bonhomme que je ne savais pas dessiner. Il me répondit :

-Ça ne fait rien. Dessine-moi un mouton.

Le Petit Prince A. de Saint-Exupéry

Formulation. Étant donné un graphe $G = (V, E)$ (pondéré ou non), $V = \{v_1, \ldots, v_n\}$, dessiner ce graphe de façon estétique(?).

L'idée de l'algorithme est de regarder la distance euclidien entre les points représentants chaque paire de sommets de G comme la distance graphique entre ces deux sommets. Soit $p_1, \ldots, p_n$ les points du plan representants les sommets $v_1, \ldots, v_n$. Nous introduisons une système dynamique de particules en considérant $p_1, \ldots, p_n$ comme des particules mobiles et on veut minimiser l'énergie totale du système, i.e.

$$E = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{1}{2} k_{ij} (|p_i - p_j| - l_{ij})^2. \tag{1}$$

Ici $l_{ij} = L \times d(v_i, v_j)$, où L est la longueur desirable d'une arête sur l'ectan de l 'ordinateur et $d(v_i, v_j)$ est la distance entre v_i et v_j dans le graphe G. Comme valeur de L on pourra prendre le rapport $L = L_0/diam(G)$ où L_0 est la taille de l'ecran et $diam(G)$ est le diamètre du graphe G. Le paramètre $k_{ij} = k_{ji}$ est défini en prenant $k_{ij} = K/d_{ij}^2$ avec une constante K (essayer plusieurs valeurs pour K).

Soit $p_1 = (x_1, y_1), \ldots, p_n = (x_n, y_n)$ les coordonnées courrants des points. Alors E est une fonction de $2n$ variables

$$E = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{1}{2} k_{ij} [(x_i - x_j)^2 + (y_i - y_j)^2 + l_{ij}^2 - 2 l_{ij} \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}]. \tag{2}$$

Au lieu des minima globaux de la fonction multimodale E on va chercher des minima locaux en utilisant une méthode du type Newton-Raphson. Une condition nécessaire pour cela est

$$\frac{\partial E}{\partial x_m} = \frac{\partial E}{\partial y_m} = 0 \qquad \text{pour } 1 \leq m \leq n. \tag{3}$$

On peut reécrire les dérivées partielles de la façon suivante.

$$\frac{\partial E}{\partial x_m} = \sum_{i \neq m} k_{mi} [(x_m - x_i) - \frac{l_{mi}(x_m - x_i)}{((x_m - x_i)^2 + (y_m - y_i)^2)^{1/2}}] \tag{4}$$

$$\frac{\partial E}{\partial y_m} = \sum_{i \neq m} k_{mi} [(y_m - y_i) - \frac{l_{mi}(y_m - y_i)}{((x_m - x_i)^2 + (y_m - y_i)^2)^{1/2}}]. \tag{5}$$

On va déplacer une seule particule p_m à la fois, i.e. on va regarder E comme une fonction de deux variables x_m et y_m et on va chercher un minimum local de E qui vérifie les conditions (3). Pour cela on choisira la particule que maximise le gradient Δ_m défini par

$$\Delta_m = \sqrt{\left(\frac{\partial E}{\partial x_m}\right)^2 + \left(\frac{\partial E}{\partial y_m}\right)^2}.$$

En débutant par le point $(x_m^{(0)}, y_m^{(0)})$ égal à la position courante (x_m, y_m), le pas suivant est réitéré avec

$$x_m^{(t+1)} = x_m^{(t)} + \delta x, \quad y_m^{(t+1)} = y_m^{(t)} + \delta y \qquad \text{pour } t = 0, 1, 2, \ldots; \tag{6}$$

Les δx et δy vérifient le système de deux équations linéaires :

$$\frac{\partial^2 E}{\partial x_m^2}(x_m^{(t)}, y_m^{(t)})\delta x + \frac{\partial^2 E}{\partial x_m \partial y_m}(x_m^{(t)}, y_m^{(t)})\delta y = -\frac{\partial E}{\partial x_m}(x_m^{(t)}, y_m^{(t)}), \tag{7}$$

$$\frac{\partial^2 E}{\partial y_m \partial x_m}(x_m^{(t)}, y_m^{(t)})\delta x + \frac{\partial^2 E}{\partial y_m^2}(x_m^{(t)}, y_m^{(t)})\delta y = -\frac{\partial E}{\partial y_m}(x_m^{(t)}, y_m^{(t)}). \tag{8}$$

A leur tour, les coefficients de (7) et (8) sont calculés à partir des dérivées partielles (4) et (5) comme suit

$$\frac{\partial^2 E}{\partial x_m^2} = \sum_{i \neq m} k_{mi}(1 - \frac{l_{mi}(y_m - y_i)^2}{((x_m - x_i)^2 + (y_m - y_i)^2)^{3/2}}), \tag{9}$$

$$\frac{\partial^2 E}{\partial x_m \partial y_m} = \sum_{i \neq m} k_{mi} \frac{l_{mi}(x_m - x_i)(y_m - y_i)}{((x_m - x_i)^2 + (y_m - y_i)^2)^{3/2}}, \tag{10}$$

$$\frac{\partial^2 E}{\partial y_m \partial x_m} = \sum_{i \neq m} k_{mi} \frac{l_{mi}(x_m - x_i)(y_m - y_i)}{((x_m - x_i)^2 + (y_m - y_i)^2)^{3/2}}, \tag{11}$$

$$\frac{\partial^2 E}{\partial y_m^2} = \sum_{i \neq m} k_{mi}(1 - \frac{l_{mi}(x_m - x_i)^2}{((x_m - x_i)^2 + (y_m - y_i)^2)^{3/2}}). \tag{12}$$

Alors δx et δy peuvent être calculer en utilisant les expressions (7)-(12). L'itération (6) se termine si Δ_m devient suffisamment petit.

Algorithme JOLIGRAPHE

```
calculer la matrice des distances (d_ij) de G;
calculer (l_ij) et (k_ij);
initialiser les particules p_1,...,p_n (placer uniformement p_1,...,p_n sur le
cercle de diametre L_0);
while max_i Δ_i > ε do
      begin
      soit p_m la particule qui vérifie Δ_m = max_i Δ_i;
      while δ_m > ε do
            begin
           calculer δx et δy en résolvant (7) et (8) ;
           x_m := x_m + δx;
           y_m := y_m + δy;
           end ;
       end
```

Annexe II : Distances arborées

Dans cette annexe nous presentons les preuves de certains résultats sur les distances arborées et ultramétriques.

Théorème 1. *Une distance d sur S est une distance arborée si et seulement si elle vérifie la condition des quatre points :*

$$\textit{pour tous } x, y, z \in X, d(x,y) + d(z,t) \leq max\{d(x,z) + d(y,t), d(x,t) + d(y,z)\}.$$

Preuve. La démonstration est algorithmique (conseil : illustrer la preuve par un dessin). On ordonne de façon arbitraire les points de S, disons $S = \{s_1, s_2, \ldots, s_n\}$. Comme arbre T_2 on prendr un segment de longueur $d(s_1, s_2)$ avec s_1 et s_2 comme extrémités. Ayant construit l'arbre T_i qui réalise les distances entre les sommets $s_1, \ldots, s_i$, l'arbre T_{i+1} est défini de la façon suivante. Soit s_j, s_k une paire de sommets avec $j, k \leq i$ qui minimise l'écart $\frac{1}{2}(d(s_{i+1}, s_j) + d(s_{i+1}, s_k) - d(s_j, s_k))$. Pour simplifier les notations, notons $a := s_j, b := s_k$ et $x := s_{i+1}$. Pour le sommet courrant $x = s_{i+1}$ soit

$$d_x = 1/2(d(x,a) + d(x,b) - d(a,b)),$$

$$d_a = 1/2(d(a,x) + d(a,b) - d(x,b))$$

$$d_b = 1/2(d(b,x) + d(b,a) - d(x,a)).$$

Sur le chemin P de T_i entre a et b (considéré comme un segment continu) on choisit le point s situé à distance d_a de a et à distance d_b de b. Ensuite, on ajoute a l'arbre T_i l'arête (s, x) de longueur d_x. L'arbre resultant s'appelle $T := T_{i+1}$. Nous affirmons que pour n'importe quel sommet $y := s_l, l \leq i$, on a $d(s_{i+1}, s_l) = d_T(x, y)$. On peut vérifier facilement cette égalite dans le cas $y = a, b$.

Maintenant supposons que $y \neq a, b$. Soit t le sommet le plus proche de y du chemin P. Supposons sans perte de généralité que t est situé sur la portion de P entre s et b (on peut bien avoir $t = b$ ou $t = s$). Notons $\lambda = d_T(s,t), d_y = d_T(y,t), d'_b = d_T(t,b) = d_b - \lambda$. On sait déjà que

$$d(a,b) = d_T(a,b) = d_a + \lambda + d'_b, d(a,x) = d_T(a,x) = d_a + d_x, d(b,x) = d_T(b,x) = d'_b + \lambda + d_x, \tag{1}$$

$$d(a,y) = d_T(a,y) = d_a + \lambda + d_y, d(y,b) = d_T(y,b) = d_y + d_b. \tag{2}$$

À partir de la condition des quatre points on va distinguer trois cas (rappelons que $a = s_j, b = s_k, x = s_{i+1}, y = s_l$).
Cas 1. $d(x,y) + d(a,b) = d(a,x) + d(y,b) \geq d(x,b) + d(a,y)$.
En utilisant (1) et (2) on déduit :

$$d(x,y) + d_a + \lambda + d'_b = d_a + d_x + d_y + d'_b \geq d_x + \lambda + d_b + d_a + \lambda + d_y.$$

L'inégalité implique que $\lambda = 0$, donc dans ce cas $d(x,y) = d_x + d_y = d_T(x,y)$.
Cas 2. $d(x,y) + d(a,b) \leq d(a,x) + d(y,b) = d(a,y) + d(x,b)$.
Encore une fois, de (1) et (2) on déduit :

$$d(x,y) + d_a + \lambda + d'_b \leq d_a + d_x + d_y + d'_b = d_a + \lambda + d_y + \lambda + d'_b,$$

donc $\lambda = 0$. Si $d(x,y) < d_x + d_y$, alors $\frac{1}{2}(d(x,a) + d(x,y) - d(a,y)) < d_x$, en contradiction avec le choix de la paire a, b.
Cas 3. $d(x,y) + d(a,b) = d(x,b) + d(a,y)$.

Dans ce cas $d(x,y) = d_x + \lambda + d'_b + d_a + \lambda + d_y - d_a - \lambda - d'_b = d_x + d_y + \lambda = d_T(x,y)$ □

Annexe III : La couverture de clique d'un graphe triangulé

On va commencer par une propriété caractéristique des graphes triangulés. Un sommet *simplicial* d'un graphe est un sommet dont les voisins sont deux à deux adjacents (i.e. ils forment une clique). Un ordre total $v_1, v_2, \ldots, v_n$ des sommets d'un graphe $G = (V, E)$ avec n sommets s'appelle un *ordre parfait* de G si le sommet v_i est un sommet simplicial dans le sous-graphe G_i de G induit par l'ensemble $V_i := \{v_1, v_2, \ldots, v_i\}$. Soit $N_u = \{v_j \in V_i : (u, v_j) \in E\}$. Il est clair que si G admet un ordre parfait alors G est triangulé : si G contient un circuit C de longueur ≥ 4 sans corde, dans n'importe quel ordre total des sommets de G le sommet v_j de C avec le plus grand numéro ne sera pas un sommet simplicial du graphe G_j. D'autre part, il est bien connu que chaque graphe triangulé admet un ordre parfait. On va présenter une preuve algorithmique, notamment on va montrer q'un ordre légal d'un graphe triangulé G est un ordre parfait. Rappelons, que dans un ordre légal de G à l'itération i le sommet v_i est le sommet non numéroté avec un nombre maximum de voisins déjà numérotées.

Théorème 1. *Soit $v_1, v_2, \ldots, v_n$ est un ordre légal d'un graphe triangulé G. Alors $v_1, \ldots, v_n$ est un ordre parfait de G.*

Preuve. On appelle un sous-ensemble des sommets S de V *convexe* si pour n'importe quels sommets $u, v \in S$ tous les (u, v)-chemins sans cordes appartiennent à S.

Lemme 1. *Chaque V_i est convexe.*

Preuve du Lemme 1. Par induction, en partant du cas trivial $i = 1$. Supposons que V_{i-1} est convexe, mais V_i ne l'est pas. Alors on peut choisir un sommet v_j non adjacent à v_i, $j < i$, est un (v_i, v_j)-chemin $P = (v_i, u_1, \ldots, u_k, v_j)$ tel que $P \cap V_i = \{v_i, v_j\}$. Soit v_l n'importe quel voisin de v_i dans V_i. Puisque $v_l, v_j \in V_{i-1}$ et que l'ensemble V_{i-1} est convexe par hypothèse d'induction, les sommets v_l et v_j doivent être adjacents. Le chemin P et les arêtes $(v_i, v_l), (v_l, v_l)$ constituent un circuit C de longueur ≥ 4. Par le choix du chemin P, on peut facilement déduire de la triangularité de G que le sommet v_l doit être adjacent à tous les sommets de P, en particulier v_l et u_k sont adjacents. Puisque v_l est un sommet arbitraire de N_{v_i} on déduit que le sommet u_k est adjacent à tous les sommets de N_{v_i}. Comme v_k est adjacent à v_j et v_i ne l'est pas, ça contredit le fait que le sommet v_i précède v_j dans l'ordre légal. $\square$

Comme V_{n-1} est convexe, tous les voisins du sommet v_n doivent être adjacents, donc v_n est un sommet simplicial de G. De façon analogue, on montre que chaque sommet v_i est simplicial dans G_i. $\square$

Soit $v_1, v_2, \ldots, v_n$ un ordre parfait du graphe triangulé G. On va construire en parallèle une couverture par des cliques et un stable (un sous-ensemble de sommets deux à deux non-adjacents) de G ayant la même cardinalité. Comme n'importe quelle clique d'une couverture par des cliques $\mathcal{C}$ peut contenir au maximum un sommet d'un stable S, on a $|S| \leq |\mathcal{C}|$. D'ici on obtient que la couverture de clique construit est de taille minimum et que le stable construit est de taille maximum. On construit le stable $S = \{s_1, s_2, \ldots, s_t\}$ de façon récursive, en effectuant un parcours inverse de la liste $v_1, \ldots, v_n$: soit $s_1 = v_n$; s_i est le premier sommet de la liste $v_{n-1}, v_{n-2}, \ldots, v_2, v_1$ après s_{i-1} et qui n'appartient pas à l'union $N_{s_1} \cup N_{s_2} \cup \cdots \cup N_{s_{i-1}}$. La couverture par des clique $\mathcal{C} = \{Q_1, \ldots, Q_t\}$ est définie comme suit : $Q_i = \{s_i\} \cup N_{s_i}$ $i = 1, \ldots, t$. La définition de S implique que S est bien un stable. D'autre part, le Théorème 1 implique que chaque Q_i est une clique, donc $\mathcal{C}$ est une couverture par des cliques du graphe G. Puisque $|S| = |\mathcal{C}|$, on obtient le résultat suivant.

Théorème 2. *S est un stable maximum et $\mathcal{C}$ est une couverture minimum par des cliques du graphe G.*