MACABRE Loona
RANAIVO Harimbola

2022 - 2023
COMPLEX

# Rapport de projet

## Coupe minimum dans un graphe et algorithmes probabilistes

Ce projet a été réalisé en python. Pour tout le projet, nous représenterons les sommets d'un graphe par un entier allant de 1 à n.

**Partie 1 - Algorithme de Karger**

- **Première structure de donnée : Matrice d'adjacence**

Pour tester nos algorithmes, nous avons implémenté les graphes avec des tableaux. Dans un premier temps, nous représentons un graphe comme une matrice d'adjacence, le tableau correspondant est donc de la forme `[matrice, n, E]`, où `matrice` est la matrice d'adjacence, `n` le nombre de sommets du graphe, et `E` la liste de ses arêtes. `E` est une liste de tuples `(i,j)` où `(i,j)` est une arête du graphe. Comme les graphes sont non orientés, pour toute arête `e = (i,j)` appartenant à E, nous considérerons `i<j`. Par exemple, une arête entre les sommets 3 et 6 sera de la forme `(3, 6)`.

- **Contraction d'une arête**

Maintenant que les structures de données représentant les graphes sont définies, nous pouvons passer à l'implémentation de l'algorithme de Karger. Nous commençons par implémenter la fonction `contract(M,e)` réalisant l'opération de contraction d'une arête `e` dans la matrice d'adjacence `M`. La contraction d'une arête `e = (i, j)` se déroule comme suit: on considère les colonnes des deux sommets concernés par la contraction, on additionne ces deux colonnes et, par convention, on stocke le résultat dans la colonne d'indice le plus petit, c'est-à-dire i. On copie ensuite les données de la colonne i sur la ligne i. Dans la colonne de j, la première ligne contient l'opposé de i, pour indiquer que ces deux sommets partagent une arête contractée. La première ligne permet donc de reconnaître les sommets résultant d'une contraction si celle-ci contient un entier négatif. Par exemple, dans la matrice d'adjacence suivante:

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \end{pmatrix}$$

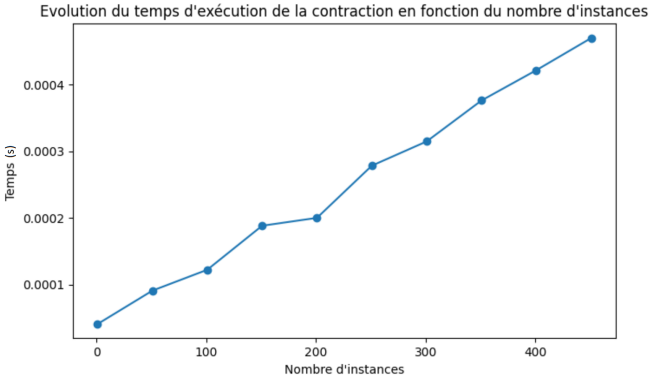
Si on contracte l'arête (2, 3), cela donne:

$$\begin{pmatrix} 0 & 1 & -2 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \end{pmatrix}$$

L'opération de contraction est en temps linéaire avec les matrices d'adjacence:

- L'accès aux colonnes d'indices i et j se fait en temps constant
- L'addition se fait en O(n) car on parcourt chaque ligne de la matrice

Nous avons testé notre fonction de contraction sur diverses familles de graphes, notamment des graphes complets et bipartis complets, des graphes cycliques et des graphes aléatoires. Pour vérifier que la complexité de l'opération de contraction est bien linéaire, nous avons calculé le temps d'exécution (en secondes) sur des instances de graphes aléatoires de plus en plus grandes et collecté les résultats sous forme de graphiques en espérant voir se dessiner une droite dans le plan (nombre d'instances, temps):



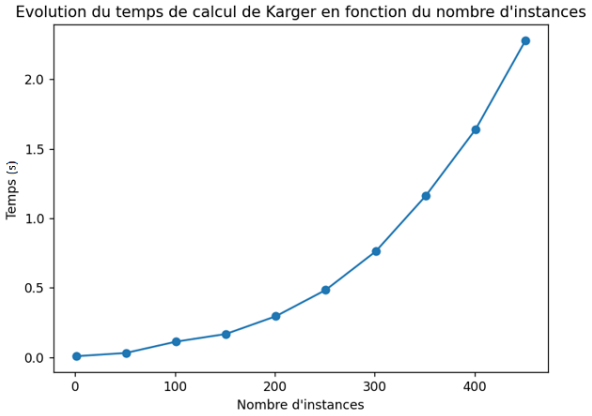

- **Implémentation de Karger**

Nous passons ensuite à l'implémentation de l'algorithme de Karger sur nos graphes. Pour cela nous créons une méthode `karger()` dans laquelle à chaque itération nous tirons aléatoirement une arête de `E` à l'aide de la méthode `random.choice()` de la bibliothèque `random` en python, prenant en paramètre un objet itérable (ici, la liste `E`). D'après nos recherches, `random.choice()` semble appeler en premier la méthode `random.randint()` (ou équivalent) qui tire un entier aléatoire `i uniformément` sur la taille de la liste en temps constant, puis accède en temps constant également à l'élément d'indice `i`. La sélection de l'arête se ferait donc en O(1). Cependant, plusieurs autres sources indiquent une complexité en la taille de la liste… soit du O(m), où m est le nombre d'arêtes du graphe. Ce n'est pas ce qu'on souhaite, mais faute de temps et de meilleure solution, nous sommes restés sur cette implémentation.

La complexité de notre algorithme dépend de celle de `contract()`, qu'il appelle à chaque itération sur le nombre de sommets. On s'arrête lorsqu'il ne reste que 2 sommets, donc on itère au maximum n - 2 fois. `contract()` a une complexité en O(n), `karger` a donc une complexité en $O(n^2)$, ce qui est cohérent avec la complexité théorique vue en cours. Nous avons aussi une boucle dans notre algorithme qui nous permet d'accéder aux sommets contenus dans les sous-ensembles de la coupe à l'issue des opérations de contraction : cette étape se fait en O(n).

A la fin de la boucle de `karger`, la première ligne de la matrice d'adjacence ne contient plus que 2 colonnes à entiers positifs, ces 2 colonnes représentent les 2 sommets restants. Pour obtenir un ensemble S de la coupe, on se base sur une propriété de l'implémentation. En effet, comme à chaque contraction on stocke le résultat dans le plus petit indice, le sommet 1 sera toujours un des deux sommets restants, donc on ajoute 1 à S directement. Il suffit alors de parcourir la première ligne de la matrice et de tester si le sommet *s* fait partie de la contraction du sommet 1, si oui, l'ajouter à S.

Nous testons notre algorithme de Karger sur les instances de graphe décrites plus tôt. Nous remarquons qu'il se trompe assez rarement, voire jamais pour certaines familles de graphes (comme les graphes cycliques ou complets). Comme précédemment, nous avons dessiné une courbe permettant de visualiser la complexité temporelle de notre méthode en fonction du nombre de sommets d'un graphe aléatoire:

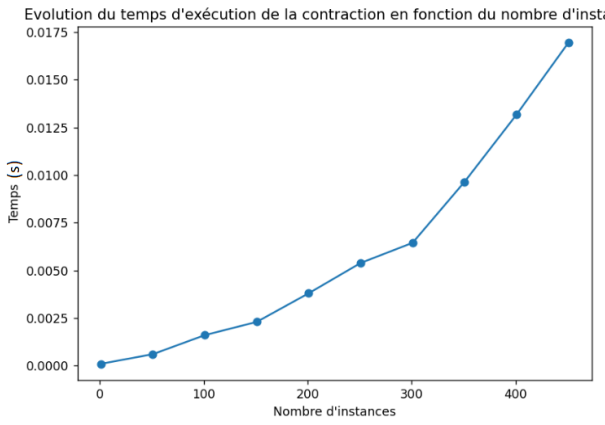



On reconnaît bien là une fonction du second degré. Cela nous rassure sur l'impact du tirage aléatoire dans les arêtes.

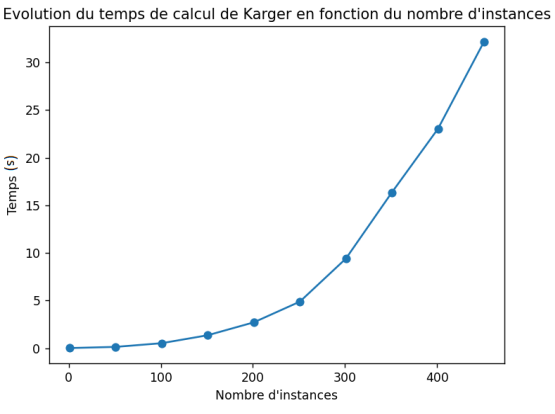
- **Deuxième structure de donnée: listes d'adjacence**

Nous avons ensuite choisi d'implémenter les graphes avec des listes d'adjacence. Les graphes sont toujours sous la forme de tableaux: `[listes, E]`, où `listes` est une liste de listes `[L1, L2, … Ln]` avec `n` le nombre de sommets, et où `Li` est la liste des sommets adjacents au sommet `i`. `E` est la liste des arêtes (identique à l'implémentation précédente). Nous adaptons alors toutes les méthodes que nous avons créées pour les matrices: `contractListe()` et `kargerListe()` ainsi que les fonctions générant différentes familles de graphes. Nous n'avons pas réussi à conserver une complexité linéaire pour l'opération de contraction: pour une arête `(i,j)` où `i<j`, le principe est de stocker la liste d'adjacence du sommet `j` dans celle du sommet `i`. Ensuite, nous remplaçons toutes les occurrences de `j` dans les listes d'adjacence de tous les sommets, pour indiquer que ce sommet "n'existe plus" dans le graphe et que toutes les arêtes menant vers ce sommet mènent dorénavant vers `i`. Pour cela, nous parcourons chaque indice de `listes`, soit n itérations, et pour chaque sommet on parcourt sa liste d'adjacence afin de trouver `j` et de le remplacer par `i`. On a donc une complexité en $O(n^2)$. `karger` ne change pas vraiment entre cette implémentation et la précédente, on itère `n - 2` fois les appels à `contractListe`, ce qui fait $O(n^3)$. C'est bien ce que l'on observe ici, sur des instances de graphes aléatoires:

`contractListe`



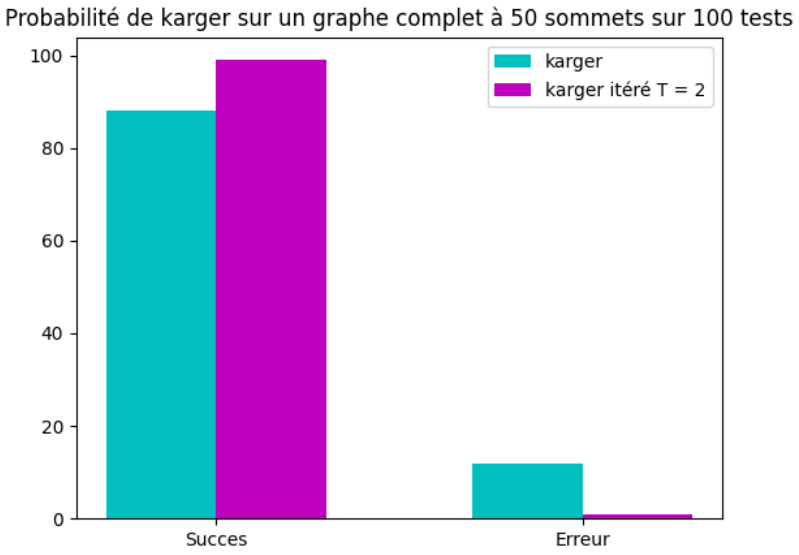

`kargerListe`



L'implémentation avec les matrices d'adjacence est donc plus performant, c'est celui que nous allons utiliser pour la suite du projet.

## Partie 2 - Algorithme de Karger itéré

Pour diminuer la probabilité d'échec de l'algorithme, on va exécuter l'algorithme de Karger T fois et prendre le minimum des cardinaux que celui-ci retourne. Nous créons pour cela la méthode `karger_itere()`. Nous allons tester l'évolution du nombre de bonnes réponses de l'algorithme en fonction de T. Nous prenons un graphe complet à 50 sommets, la coupe minimum est de cardinal n - 1 = 49. Nous allons comparer le taux de succès de karger avec celui de karger itéré:




On observe bien que l'algorithme itéré a un taux d'erreur plus faible ce qui nous permet d'affirmer qu'il est plus performant. A chaque itération on repart sur le graphe originel. On itère T fois sur karger qui a une complexité $O(n^2)$, donc ici la complexité serait $O(n^2 * T)$. Si on veut maximiser la probabilité de succès, on a intérêt à augmenter T, ce qui augmente aussi la complexité. Lorsque T est supérieur ou égale à n, on aura au mieux une complexité cubique.

D'après nos tests sur un même nombre de sommets de plusieurs familles de graphe, pour un T assez élevé, l'algorithme de karger itéré a quasi 100% de chance de retourner une coupe minimum. Mais en contrepartie on observe une hausse du temps d'exécution.

## Partie 3 - Algorithme de Karger-Stein

Cet algorithme est un algorithme récursif de type "Diviser pour régner".
Il nous permet d'améliorer le taux de réussite pour la recherche de coupe minimum. De plus, la complexité est inférieure à celle de karger itéré. Le déroulement est le suivant: on fait 2 copies du graphe courant, et on contracte des arêtes aléatoirement jusqu'à ce qu'il ne reste plus que $n/\sqrt{2}$ sommets sur ces deux graphes. Cette opération est réalisée par la fonction de

contraction partielle. Puis on appelle récursivement l'algorithme de Karger-Stein sur ces deux nouveaux graphes. La condition d'arrêt est lorsqu'il ne reste plus que 6 sommets, dans ce cas là on appelle la méthode `karger_itere()` pour obtenir une coupe. On garde alors à chaque fois la coupe minimale entre le retour des 2 graphes.

Montrons le temps d'exécution d'un tel algorithme:

On considère un graphe G auquel on applique à deux reprises n - $\frac{n}{\sqrt{2}}$ - 1 contractions aléatoire jusqu'à ce qu'il ne reste plus que $\frac{n}{\sqrt{2}}$ + 1 sommets dans les deux cas. On obtient donc deux nouveaux graphes $G_1$ et $G_2$ dont le nombre de sommets est $n_1 = n_2 = 1 + \frac{n}{\sqrt{2}}$. On appelle récursivement l'algorithme sur ces deux graphes. Le temps de calcul sur n sommets vaut donc le temps dû aux contractions partielles en $O(n^2)$ additionné au temps de calcul sur les deux graphes, soit:

$$T(n) = 2T(\lceil 1 + \frac{n}{\sqrt{2}} \rceil) + O(n^2)$$

Le nombre total de sommets est divisé par $\sqrt{2}$ à chaque récursion, ce qui permet de déduire que $T(\lceil 1 + \frac{n}{\sqrt{2}} \rceil)$ représente une complexité logarithmique. On obtient donc:

$$T(n) = O(n^2 \log n)$$

Calculons la probabilité de succès de l'algorithme de Karger-Stein:

Prenons le graphe $G$ dont une coupe minimum est $C$. La probabilité de succès de l'algortihme dépend de la contraction partielle: l'algorithme retourne $C$ si à chaque étape $i$, l'arête $e_i$ n'appartient pas à $C$. Si on note $E_i$ l'événement "l'arête sélectionnée n'appartient pas à $C$", on a:

$$P(\bar{E}_1) = \frac{|C|}{|E|}$$

Tout sommet a au moins $|C|$ arêtes adjacentes, donc $|E| \geq |C| \times \frac{n}{2}$ donc

$$P(E_1) = 1 - \frac{|C|}{|E|} \geq 1 - \frac{|C| \times 2}{|C| \times n} = \frac{n-2}{n}$$

A l'étape $i$, il reste $n - i + 1$ sommets dans le graphe donc $|E|_i >= |C| \times \frac{(n-i+1)}{2}$. On a alors:

$$P(E_i|E_1, E_2, ...E_{i-1}) = 1 - \frac{|C|}{|E|_i} \geq 1 - \frac{|C| \times 2}{|C| \times (n-i+1)} = \frac{n-i-1}{n-i+1}$$

La probabilité de l'événement E: ne séléctionner que des arêtes ne traversant pas $C$ est:

$$\begin{aligned}
P(E) &= P\left(\bigcap_{i=1}^{n-\frac{n}{\sqrt{2}}-1} E_i\right) \\
&= P(E_1)P(E_2|E_1)...P(E_{n-\frac{n}{\sqrt{2}}-1}|E_{n-\frac{n}{\sqrt{2}}-2}) \\
&\geq (\frac{n-2}{n})(\frac{n-3}{n-1})...(\frac{\frac{n}{\sqrt{2}}}{\frac{n}{\sqrt{2}}+2}) \\
&= \frac{\frac{n}{\sqrt{2}}(\frac{n}{\sqrt{2}}+1)}{n(n-1)} \\
&\geq \frac{1}{2}
\end{aligned}$$

Si on pose $C_1$ la coupe retournée par l'algortihme pour le graphe $G_1$, $C_2$ la coupe retournée par l'algorithme pour le graphe $G_2$, alors $C_1$ est une coupe minimum de $G$ si elle est une coupe minimum de $G_1$ et que l'appel récursif à la contraction de $G_1$ retourne une coupe minimum de $G_1$. On peut alors dire:

$$\begin{aligned} P(C_1 \text{ est une coupe min de } G) &= P(C_1 \text{ est une coupe min de } G_1, \text{ l'appel récursif retourne une coupe min de } G_1) \\ &= P(C_1 \text{ est une coupe min de } G_1) \times P(\text{ l'appel récursif retourne une coupe min de } G_1) \\ &\geq \frac{1}{2} P\left(\frac{n}{\sqrt{2}}\right) \end{aligned}$$

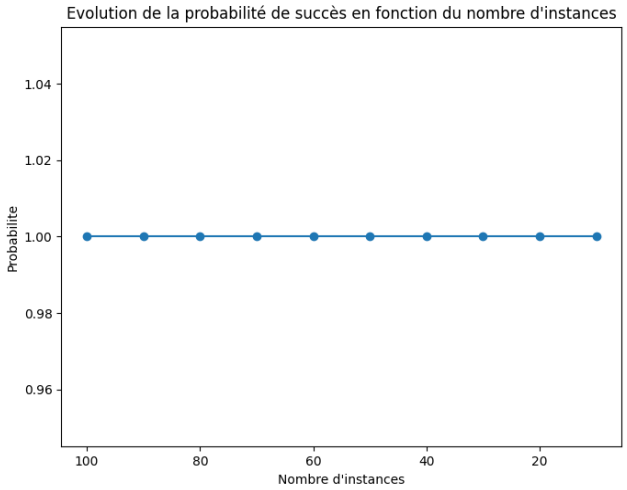
Donc, $P(C_1/C_2 \text{ n'est pas une coupe min de } G) \leq 1 - \frac{1}{2} P\left(\frac{n}{\sqrt{2}}\right)$

On a finalement:

$$\begin{aligned} P(\text{échec}) &= P(C_1 \text{ n'est pas une coupe min de } G, C_2 \text{ n'est pas une coupe min de } G) \\ &= P(C_1 \text{ n'est pas une coupe min de } G) \times P(C_2 \text{ n'est pas une coupe min de } G) \\ &\leq \left(1 - \frac{1}{2} P\left(1 + \frac{n}{\sqrt{2}}\right)\right)^2 \end{aligned}$$

D'où $P(n) \geq 1 - \left(1 - \frac{1}{2} P\left(1 + \frac{n}{\sqrt{2}}\right)\right)^2$

Nous avons tenté d'étudier la probabilité de succès de notre algorithme, cependant lorsque nous traçons l'évolution de la probabilité de succès de `karger_stein()` sur 100 sommets, nous obtenons la courbe suivante:




Ce test a été fait sur des graphes cycliques dont on connaît la coupe minimale, qui est 2. Comme nous l'avons évoqué plus tôt, notre implémentation de Karger ne se trompait jamais, c'est encore le cas pour Karger-Stein. P(n) est toujours à 1, donc 1/log n est bel et bien une borne inférieure de P(n) d'où P(n) = Ω(1/ log n), mais ce n'est pas très pertinent au vu de nos résultats expérimentaux.