Algorithmique (12X001) - Automne 2023

11. Algorithmes d'approximation (1)

Enseignant: Arnaud Casteigts *Assistant: Brian Pulfer*

Que faire lorsqu'on est confronté à un problème difficile ? Plusieurs options s'offrent à nous, parmi lesquelles : 1) le résoudre quand même optimalement, tant pis pour le temps de calcul ; 2) restreindre le type d'instances qu'on cherche à traiter, les cas particuliers étant moins difficiles que le cas général ; 3) abandonner l'objectif de trouver une solution optimale, tout en cherchant à ne pas trop s'en éloigner.

Les **algorithmes d'approximation** s'inscrivent dans la troisième catégorie : on veut savoir à quel point on peut s'**approcher** de la solution optimale, tout en gardant une **complexité polynomiale** en temps. Cette approche s'adresse généralement aux problèmes NP-difficiles, même si on peut la décliner à d'autres échelles.

La qualité que l'on peut atteindre varie selon les problèmes. Soit n la taille de l'entrée à traiter. Supposons que l'on a affaire à un problème de *minimisation* et appelons OPT la valeur de la solution optimale. Voici les trois niveaux d'approximabilité les plus classiques :

- $(1+\epsilon)\cdot OPT$: c'est le meilleur scénario possible pour un problème NP-difficile. On peut s'approcher ici aussi près qu'on veut de OPT, ϵ étant un paramètre à choisir. Bien sûr, ça coûtera d'autant plus cher qu'on s'en approche, mais ce sera toujours polynomial en n, p.ex. ça coûterait un temps $O(n^5)$ pour $\epsilon = 1/5$, $O(n^{12})$ pour $\epsilon = 1/12$, etc.
- $k \cdot OPT$: c'est le scénario le plus classique, où l'on arrive à garantir que notre solution n'excède jamais k fois l'optimum, pour une certaine valeur de k que l'on ne choisit pas. Par exemple, pour certains problèmes, il existe des algorithmes qui garantissent $2 \cdot OPT$, on dit qu'un tel algorithme donne une 2-*approximation*.
- inapproximable : c'est le pire scénario, où quel que soit k, il n'existe aucun algorithme qui donne une k-approximation.

La même classification s'applique pour les problèmes de maximisation, en inversant le facteur. Par exemple, une k-approximation garantit une solution de qualité OPT/k.

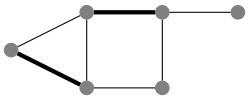
Il existe des subdivisions supplémentaires, mais ces notions sont suffisantes pour débuter. Classifier les problèmes NP-difficiles en fonction de leur approximabilité est un sujet de recherche encore actif, qui a connu de nombreux développements au cours des trente dernières années.

11.1 Deux problèmes emblématiques

Nous allons discuter de deux problèmes qui sont souvent donnés en exemples sur le sujet. Ce sont tous deux des problèmes de graphes.

11.1.1 Couplages (et discussions)

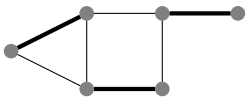
Soit un graphe $G = (V, E)$. Un **couplage** dans G (*matching*, en anglais) est un ensemble d'arêtes $E' \subseteq E$ qui ne se touchent pas (autrement dit, ne partagent aucun sommet). Voici quelques exemples :



Note : l'ensemble $E' = \emptyset$ est un couplage valide, mais pas très intéressant. On cherche généralement à **maximiser** la taille d'un couplage. Il y a deux notions de maximalité qu'il est important de distinguer : solution *maximale* ou solution *maximum*. Les deux ne sont pas les mêmes :

- Maximale = On ne peut rien lui ajouter
- Maximum = Il n'en existe pas de plus grande.

Ce n'est réellement pas pareil, par exemple, la solution précédente était maximale, car si on lui ajoute n'importe quelle autre arête, ce n'est plus un matching. Mais elle n'est pas maximum, car il on peut trouver un couplage avec plus d'arêtes, par exemple :



Ainsi, une solution maximale n'est pas forcément maximum. Par contre, une solution maximum est toujours maximale. Naturellement, trouver une solution maximum est plus difficile, pour n'importe quel problème d'optimisation. À l'inverse, en trouver une maximale est toujours facile, car il suffit d'utiliser un algorithme glouton.

Lorsqu'on parle de solution *optimale* à un problème d'optimisation, malgré le suffixe "-al", on se réfère bien à une solution maxim**um**. En fait, optimal et optimum sont ici synonymes (gare aux confusions !).

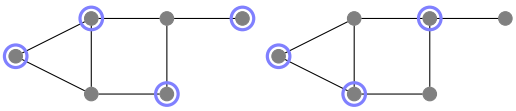
Complexité ?

- Minimal Matching : polynomial
- Minimum Matching : polynomial

Montrer que Minimum Matching est polynomial pourrait faire l'objet d'un cours entier ! Mais ce n'est pas le sujet aujourd'hui.

11.1.2 Couverture par sommets

Soit un graphe $G = (V, E)$. Une **couverture par sommets** dans G (*vertex cover*, en anglais) est un ensemble de sommets $V' \subseteq V$ qui (collectivement) touchent toutes les arêtes. Par exemple :



Une solution triviale ici est $V' = V$ (valide, mais pas très intéressant). En général, on cherche à *minimiser* la taille d'une solution. On peut à nouveau distinguer la différence entre une solution qui est seulement *minimale* (comme celle de gauche) et une solution qui est *minimum* (celle de droite).

Complexité ?

- Minimal Vertex Cover : polynomial
- Minimum Vertex Cover : NP-difficile !

Et oui, pas de chance, trouver une couverture optimale est un problème difficile. Voyons donc si on peut l'approximer !

11.2 Minimum Vertex Cover est 2-approximable

Supposez que l'on dispose d'une fonction qui calcule un couplage maximal quelconque (problème facile, pouvant être résolu par un algorithme glouton). On peut l'utiliser pour obtenir facilement une couverture par sommet, comme suit :

```
vertex_cover(G) :
    S ← ∅
    E' ← minimal_matching(G)
    Pour chaque arête uv de E' :
        Ajouter u à S
        Ajouter v à S
    Renvoyer S
```

Il se trouve que cet algorithme est une 2-approximation pour Minimum Vertex Cover. Commençons par vérifier que la solution renvoyée est une couverture par sommets valide.

Lemme 11.1. *Chaque arête de G est couverte par au moins un sommet de S*

Preuve. Par l'absurde, supposons qu'il existe une arête uv qui n'est pas couverte. Cela implique que ni u ni v ne touche une arête du couplage utilisé (sinon, au moins l'un d'eux

aurait été sélectionné). Mais alors, on aurait pu ajouter l'arête uv à ce couplage, qui n'était donc pas maximal (contradiction). □

Notons OPT la taille de la solution optimale pour le problème de couverture par sommets que l'on cherche à résoudre.

Lemme 11.2. $|S| \leq 2 \cdot OPT$

Preuve. Par définition, une couverture par sommets doit couvrir toutes les arêtes de G. Elle doit donc, en particulier, couvrir toutes les arêtes d'un couplage maximal E'. Mais ces arêtes étant disjointes, il faut au moins un sommet distinct pour couvrir chacune. Autrement dit, $OPT \geq E'$. Via l'algorithme, on a $|S| = 2 \cdot |E'|$, on a donc $|S| = 2 \cdot |E'| \leq 2 \cdot OPT$. □

Conclusion : Il existe une 2-approximation pour Minimum Vertex Cover. De plus, l'algorithme correspondant est un algorithme essentiellement glouton (très rapide).

Nous avons exploité ici un exemple de **dualité** entre deux problèmes, à savoir :

$$|\,\texttt{Maximal Matching}\,| \;\leq\; 2\times |\,\texttt{Minimum Vertex Cover}\,|$$

Beaucoup d'algorithmes d'approximation (mais pas tous) sont basées sur ce type de relations entre deux problèmes, l'un étant facile à résoudre et l'autre non.

11.3 Sac à dos (Knapsack)

Dans le problème du sac à dos (*knapsack*, en anglais), une instance consiste en un ensemble de n objets qui ont chacun une valeur et un poids. L'objectif est de trouver un sous-ensemble dont la valeur totale est maximum sans dépasser un poids total donné.

Exemple d'instance : l'ensemble de couples $(valeurs, poids)$ suivants : $\{(100, 20), (120, 30), (10, 10), (60, 10)\}$ ainsi que le poids total à ne pas dépasser, p. ex. 50.

Nous avons déjà vu un algorithme glouton pour résoudre ce problème rapidement, mais *non-optimalement* : Commencer par trier les couples par ratio valeur/poids décroissants. Puis, à chaque étape, choisir l'objet qui maximise le rapport valeur/poids sans excéder la limite de poids.

On peut adapter simplement cet algorithme pour garantir une 2-approximation.

Algorithme de 2-approximation. Commencer par nettoyer l'instance de départ pour enlever tous les objets qui dépassent à eux seuls le poids total autorisé (ces objets ne servent à rien). On applique ensuite l'algorithme glouton comme indiqué précédemment, mais cette

fois, on s'arrête dès qu'un objet x dépasse le seuil. À ce moment là, notre solution gloutonne cumule une certaine valeur V. L'objet refusé a lui-même une valeur v_x. Si $v_x < V$, on prend la solution gloutonne (en l'état), sinon, on prend l'objet x seul.

Pourquoi cela donne-t-il une 2-approximation ?

Déjà, réaliser que OPT est forcément inférieur à $V+v_x$. Pourquoi ? Parce-que l'algorithme choisit toujours les objets ayant le plus grand rapport valeur/poids. La solution $V + v_x$ a donc une "densité de valeur" supérieure ou égale à la solution optimale :

$$V + v_x \geq OPT$$

Par ailleurs, l'option la plus grande entre V et v_x vaut nécessairement au moins la moitié de $V+v_x$:

$$\max\{V, v_x\} \geq (V + v_x)/2$$

On a donc :

$$\max\{V, v_x\} \geq (V + v_x)/2 \geq OPT/2$$

11.4 Nommer les choses

Un algorithme qui permet de s'approcher autant qu'on veut de l'optimal (en payant la qualité polynomialement comme discuté au début du cours) est appelé un PTAS (*polynomial-time approximation scheme*, en anglais). Les problèmes qui admettent un tel algorithme définissent la classe APX. Certains problèmes n'admettent pas de tels algorithmes, ils sont alors APX-hard, ce qui signifit qu'on peut, au mieux, espérer trouver une k-approximation.

Pour montrer qu'un problème est APX-hard, on procède généralement par réduction, comme pour montrer qu'un problème est NP-hard. Ici, cependant, on a besoin d'une réduction plus spécifique, qui préserve le ratio d'approximation lors de la transformation. Par exemple, on prend une instance d'un problème X (que l'on sait APX-hard) et on la transforme en une instance du problème Y, en montrant que si on arrive à approximer cette dernière, cela nous donne une approximation pour la première. Pouvoir approximer Y contredirait donc le fait que X est APX-hard, d'où le fait que Y est aussi APX-hard.