Algorithmique Avancée

Partie 1 - Graphes

Nicolas Bousquet

August 26, 2022

Dans cette partie du cours, on parlera surtout d'algorithmes de graphes et en particulier:

- Techniques algorithmiques classiques dont algorithmes gloutons, algorithmes d'approximation et algorithmes dynamiques,
- Qualité des solutions renvoyées par les algorithmes,
- Temps d'exécution des algorithmes.

Le but est également d'illustrer des techniques de preuve classiques. Entre autres:

- Preuves par l'absurde,
- Preuves par récurrence.

Les preuves seront utilisées dans deux buts:

- Pour à la fois prouver la validité des algorithmes utilisés (i.e. montrer que l'algorithme a bien le comportement souhaité et ne fait pas n'importe quoi),
- Donner des bornes sur les temps d'exécution de l'algorithme (i.e. garantir que l'algorithme est efficace).

Note: Tout, en informatique, est une question de compromis entre performance (temps d'exécution) et qualité de la solution. Avant de concevoir un algorithme il est important de savoir en "combien de temps" vous désirez la solution. Les techniques que vous pourrez utiliser seront différentes si vous souhaitez une réponse instantanée ou si vous êtes prêts à attendre plusieurs jours le résultat !

1 Définitions et notations

1.1 Définition usuelles

Un *graphe* $G = (V, E)$ est composé d'un ensemble $V(G)$ (ou simplement V) de *sommets* et d'un ensemble E de paires d'éléments de V appelées *arêtes*. En d'autres termes, un graphe est un ensemble de "points" reliés par des "traits". Un graphe est *pondéré* si on se donne également une fonction f de poids sur les arêtes, i.e. une fonction $f : E \rightarrow \mathbb{R}$ qui associe à chaque arête un poids.

Tout au long de ce cours, on notera par $n(G)$ (ou n quand G est clair) le nombre de sommets, i.e. $n = |V(G)|$ ¹ et m le nombre d'arêtes de G .

Soit v un sommet de G , i.e. $v \in V(G)$. Un sommet w est un *voisin* de v (ou w est *adjacent* à v) si vw est une arête. L'ensemble des voisins de v se note $N(v)$ et le *degré* de v , noté $d(v)$, est la taille de $N(v)$.

Exercice 1. Pour tout graphe G , montrer que $\sum_{v \in V} d(v)$ est pair.

---

¹Pour un ensemble X , $|X|$ dénote la taille de X .

Un *chemin* est une suite $v_1, v_2, \dots, v_{k+1}$ de sommets sans répétition tel que pour tout $i \leq k$ $v_i v_{i+1}$ est une arête. Si, en plus $v_{k+1} v_1$ est une arête alors $v_1, v_2, \dots, v_{k+1}$ est un *cycle*. La *longueur* d'un chemin est son nombre de sommets moins 1. Autrement dit la longueur du chemin $v_1, v_2, \dots, v_{k+1}$ est k . La distance entre deux sommets u et v est la longueur d'un plus court chemin si un chemin existe (ou $+\infty$ si un tel chemin n'existe pas).

Un graphe G est *connexe* s'il existe un chemin entre n'importe quelle paire de sommets de G . Le *diamètre* d'un graphe est la distance maximum entre une paire de sommets de v . Une *composante connexe* est un sous ensemble de sommets X tel que (i) il n'y a pas d'arêtes entre X et $V \setminus X$ et (ii) pour tout $Y \subseteq X$, il existe une arête entre Y et $V \setminus Y$.

Exercice 2. Montrer qu'une composante connexe de G est un ensemble maximal par inclusion de sommets qui sont deux à deux à distance fini.

1.2 Quelques familles de graphes importantes

Une grille de taille $n \times m$ est un graphe tel que les sommets sont indexés par des paires (i, j) avec $i \leq n$ et $j \leq m$ et (i, j) est adjacent à (i', j') ssi $|i - i'| + |j - j'| = 1$.

Une *clique* de taille n , dénotée par K_n , est le graphe à n sommets où les sommets sont tous deux à deux adjacents.

Un graphe *biparti* est un graphe où l'ensemble de sommets peut être partitionné en deux ensembles A, B tels que toutes les arêtes ont une extrémité dans A et une extrémité dans B . Un biparti est *complet* si tous les sommets de A sont adjacents à tous les sommets de B . Le graphe complet avec des parties de tailles m et n est dénoté par $K_{n,m}$.

Question 3. • Quel est le nombre d'arêtes d'une clique à n sommets?

- Quel est le nombre d'arêtes du graphe biparti complet $K_{n,m}$?

Un *arbre* est un graphe connexe sans cycle.

Exercice 4. Montrer que:

1. Dans tout arbre, il existe un unique chemin entre chaque paire de points.
2. Tout arbre contient au moins un sommet de degré 1.
3. Tout arbre a exactement $n - 1$ arêtes.
4. En déduire qu'un arbre contient au moins deux sommets de degré 1.

Les sommets de degré 1 sont appelées les *feuilles* de l'arbre.

Une *étoile à n feuilles* $K_{1,n}$ est un graphe à $n + 1$ sommets tel que (i) il existe un sommet x incident à tous les autres sommets Y et, (ii) tous les sommets de Y ont degré 1. Notez les étoiles sont des arbres.

Une *forêt* est un graphe sans cycle (pas forcément connexe).

1.3 Représentation d'un graphe en machine

Liste d'adjacence. Les sommets sont représentés par des entiers de 1 à n . On représente le graphe avec n listes. La i -ème liste correspond à la liste des voisins du i -ème sommet v_i . Autrement dit la i -ème liste est composé de $N(v_i)$.

Avantages: Espace mémoire en $O(n + m)$.

Inconvénients: Temps d'accès à une information qui peut être de $O(n)$. Ce temps peut être réduit si au lieu de listes on crée des structures à l'accès plus rapide. Idem pour le temps de modification.

Matrice d'incidence Le graphe est représenté par une matrice $n \times n$ où l'entrée (i, j) vaut 1 si et seulement si ij est une arête du graphe.

Avantages: Accès constant, édition de graphe rapide (constant).

Inconvénients: Espace quadratique (peu approprié pour les graphes épars avec beaucoup de sommets).

1.4 Temps d'exécution d'algorithmes

Quelques ordres de grandeur.

Nb. sommets	Linéaire	Quadratique	2^n	2^{n^2}
2	2	4	4	16
10	10	100	1000	$\approx 10^{30}$
10^3	10^3	10^6	10^{300}	bcp
10^6	10^6	10^{12}	10^{300000}	bcp
10^9	10^9	10^{18}	bcp	bcp

Nombre d'opération effectué par un ordinateur de bureau par seconde: $\approx 10^9$.
Taille d'un entier dans un système 64 bits - $2^{64} \approx 10^{18}$.
Nombre d'atomes dans l'univers: 10^{80} .

On voit dans le tableau ci-dessus que le simple fait de passer de linéaire à quadratique modifie drastiquement le temps d'exécution d'un algorithme. Par exemple pour un graphe à 1 million de sommet un algorithme linéaire va s'exécuter instantanément alors qu'un algorithme quadratique nécessitera de l'ordre de 1 heure pour s'exécuter. Un algorithme quadratique devient complètement hors de propos à 1 milliards de sommet où le temps d'exécution passera immédiatement à plusieurs années.

On notera qu'un algorithme exponentiel ne peut être efficace que pour des petites instances.

→ L'algorithme doit être adapté à la taille des données ! Un algorithme exponentiel nécessite rapidement un temps complètement délirant mais dans le cas de toutes petites données (quelques unités voir dizaines d'unités), il peut représenter un bon compromis temps d'exécution / simplicité de codage.

Comment calculer la complexité d'un algorithme. Assignation de variables, opération simple (addition, multiplication), modification d'une case de tableau → 1 opération.
Boucles de taille $n \rightarrow n$ fois la complexité de l'intérieur de la boucle.

2 Arbres couvrants et parcours de graphes

Soit $G = (V, E)$ un graphe connexe. Un *arbre couvrant* est un sous ensemble E' d'arêtes de E tel que le graphe $G' = (V, E')$ est un arbre. Si G est pondéré avec une fonction f alors le *poids* d'un arbre couvrant est la somme des poids de ses arêtes. Autrement dit le poids de E' est $\sum_{e \in E'} f(e)$.

Trouver un arbre est un problème central en graphes pour des questions de:

- Routage (faire transférer l'information dans un réseau à travers un sous ensemble d'arêtes.
- Sous-routine pour de nombreuses applications dont clustering, tournées...etc...

2.1 Parcours de graphes

Étant donné un arbre T , on dit que T est *enraciné* en un sommet v si tous les arêtes ab sont orientées de a vers b si a est sur l'unique chemin de a vers b . On appelle v la *racine* de l'arbre. Un noeud w de T est dit de *profondeur* i si l'unique chemin de v vers w a longueur i . On dit par convention que la racine a profondeur 0.

Soit $G = (V, E)$ un graphe arbitraire. Un parcours en largeur de G en partant de v est un arbre construit grâce à l'algorithme 1.

Vous noterez ici que la structure de L est une structure FIFO (First In / First Out). Cette structure FIFO permet d'avoir un certain nombre de propriétés. Imaginons maintenant que l'on prenne une liste informatique "classique" qui a une structure FILO (Fist In / Last Out) a-t-on le même genre de propriétés?

La réponse est à la fois oui et non. Dans le cas FILO, on remplace le "Soit w le premier sommet ajouté à L " par "dernier sommet rajouté à L ". Ce changement permet encore d'obtenir un arbre couvrant.

Procedure 1 Algorithme BFS (Breath First Search) - Parcours en largeur

---

Input: Un graphe G , un sommet v .

Output: A spanning tree of G

$L = [v]$

$T = \emptyset$

$F = \emptyset$

while $L \neq \emptyset$ **do**

  Soit w le premier sommet ajouté à L

for all voisin u de w tel que u n'est pas dans T **do**

    Rajouter wu dans F

    Rajouter u dans L

    Rajouter u à T

end for

  Supprimer w de L

end while

Renvoyer F

Procedure 2 Algorithme DFS (Depth First Search) - Parcours en profondeur

---

Input: Un graphe G , un sommet v .

Output: A spanning tree of G

$L = [v]$

$T = \emptyset$

$F = \emptyset$

while $L \neq \emptyset$ **do**

  Soit w le dernier sommet ajouté à L

if il existe un voisin u de w tel que u n'est pas dans T **then**

    Rajouter wu dans F

    Rajouter u dans L

    Rajouter u à T

else

    Supprimer w de L

end if

end while

Renvoyer F

FIFO vs FILO. Revenons un peu aux FIFO / FILO.

- FIFO: First In / First Out.
Cette structure de données correspond à une caisse de supermarché. Le premier arrivé sera le premier à passer en caisse, ensuite ce sera au tour de la personne arrivée en seconde...etc... Quand on veut implémenter une file d'attente coté serveur on doit se servir de ce type de structure qui assure un traitement équitable entre les membres.
- FILO: First In / Last Out.
C'est une "pile de papier". Quand vous ajoutez un papier sur une pile, ce papier sera le premier que vous sortirez de la liste pour le traiter. Alors qu'elle parait "injuste" d'un point de vue social, elle est très utilisée en informatique car on fait une pile de choses à faire, et les éléments les plus profonds de la pile ont besoin que les choses les dernières opérations ajoutées soient effectuées pour pouvoir elles-mêmes être effectuées.

Exercice 5. 1. Montrer que l'ensemble d'arêtes renvoyé par l'algorithme forme un ensemble acyclique.

Indication: Montrer que l'ajout d'une arête uw à F , soit u soit w ont actuellement degré 0 dans F .

2. Soit G un graphe et $v \in V$. Montrer que si G n'est pas connexe alors un DFS/BFS renvoie la composante connexe de v .
3. En déduire que si G est connexe, l'algorithme renvoie un arbre couvrant.

Le *diamètre d'un graphe* est la plus grande distance entre des paires de points du graphes. Il est parfois utile de déterminer le diamètre d'un graphe. Le *plus long chemin* est un chemin de longueur maximum d'un graphe, autrement dit, il n'existe pas chemin plus long.

Exercice 6. 1. Montrer que si on fait un BFS alors, si u est au i -ème niveau, alors $d(u, v) = i$.

Indication: Preuve par récurrence.

2. En déduire un algorithme qui calcule les distances entre toutes les paires de sommets u, v de G .
3. Soit u au i -ème niveau d'un BFS et w au j -ème niveau du même BFS. Donnez une borne supérieure et inférieure sur $d(u, w)$?
4. Donner un lien entre diamètre du graphe et profondeur maximum d'un BFS.
5. Quel est la profondeur maximum d'un sommet lorsque que l'on effectue un BFS sur une clique? Un DFS?

Conclusion

- Plusieurs algorithmes qui se ressemblent peuvent satisfaire les même "conclusion" mais pourtant avoir des comportements assez différents.
→ Une analyse plus approfondie de l'algorithme est parfois nécessaire !

2.2 Algorithme de Kruskal

Theorem 7. L'algorithme de Kruskal renvoie un arbre couvrant minimum.

Proof. (Par simplicité on va supposer que toutes les arêtes ont des poids différents. Si certains poids peuvent être égaux, le théorème est encore vrai mais la preuve est inutilement un peu plus technique).

Soit T^* un arbre de poids minimum et T l'arbre renvoyé par l'algorithme de Kruskal. On prouve par contradiction que $T = T^*$. Supposons par contradiction que $T \neq T^*$. Comme les deux arbres ont le même nombre d'arête, il existe des arêtes dans $T \setminus T^*$. Considérons l'arête e de poids maximum dans T qui n'est pas dans T^* .

La forêt $T \setminus e$ a deux composantes connexes C_1, C_2 . Donc il existe dans T^* au moins une arête avec une extrémité dans C_1 et une extrémité dans C_2 . Soit f cette arête. L'arête f aurait du être ajoutée par l'algorithme car elle ne crée aucun cycle dans $T \setminus e$ et donc aucun cycle dans la forêt partielle F au moment où elle était considérée pour être ajoutée, une contradiction avec la définition de l'algorithme. $\square$

Procédure 3 Algorithme de Kruskal

Input: Un graphe pondéré G .

Output: Un arbre couvrant de G de poids minimum.

$F = \emptyset$

$E' = E$

while E' n'est pas vide **do**

  Soit uv l'arête de poids minimum dans E'

if $F \cup \{uv\}$ est acyclique **then**

    Rajouter uv à F

end if

  Supprimer uv de E'

end while

Renvoyer F

Discutons maintenant un peu de la complexité de cet algorithme ? En particulier comment vérifier efficacement que l'ajout d'une arête est acyclique? Une façon simple de le faire est de se souvenir des composantes connexes "partielles", autrement dit les composantes connexes induites par les arêtes de F . Pour faire ça, on va maintenir un tableau CC de taille n qui, à chaque sommet lui associe le numéro de sa composante connexe.

Initialement, les composantes connexes sont numérotées de 1 à n et chacun des sommet v_i est numéroté i . Autrement dit initialement $CC[i] = i$. On va maintenir le fait que si u et v sont dans la même composante connexe alors $CC[u] = CC[v]$.

Pour maintenir ceci on va appliquer l'Algorithme 4.

Procédure 4 Algorithme Merge - Regroupement de deux composantes connexes

---

Input: Un graphe pondéré G , un vecteur CC de taille n , une arête u, v telle que $CC[u] \neq CC[v]$.

Output: Le vecteur CC où les valeurs de la composante de u et de v sont les mêmes.

$c = CC[v]$

for all $j \leq n$ **do**

if $CC[j] = c$ **then**

$CC[j] = CC[u]$

end if

end for

Renvoyer CC

Quelle est la complexité de l'algorithme Merge? On crée un tableau de taille n ($O(n)^2$ opérations). On parcourt ensuite l'ensemble des sommets: encore $O(n)$ opérations.

Donc au total la complexité est linéaire.

On peut maintenant se demander quelle est la complexité de l'algorithme de Kruskal ?

- Trier les arêtes par ordre croissant de poids: ça prend un temps $O(m \log m)$ en utilisant QuickSort par exemple.
- Mise à jour du tableau: $O(n)$ étapes à chaque fois qu'on rajoute une nouvelle arête. On met le tableau à jour $O(n)$ fois (le nombre d'arêtes dans un arbre couvrant), donc au total la complexité est quadratique en n .

Amélioration: Union-Find. On peut se demander si cette complexité quadratique est nécessaire ou si elle peut être améliorée. En fait on peut être plus malin qu'un tableau et représenter les différentes composantes connexes via un arbre plutôt qu'un tableau. A chaque composante connexe C de la forêt partielle, on associe un arbre S enraciné dont les sommets sont les sommets de C . On crée un tableau de taille n tel que, pour chaque sommet v , le label de chaque sommet v est l'ancêtre de v dans S si v n'est pas la racine de sa composante et son label est v sinon.

---

²On rappelle que $O(n)$ signifie au plus Cn pour une constante C .

- Exercice 8.**
- (Find) Montrer que l'on peut vérifier si v et w sont dans la même composante connexe en un temps qui est la profondeur maximum des sous arbres contenant v et w .
  - (Union) Supposons que v et w soient dans des composantes différentes. Proposer un algorithme pour faire l'union de la composante de v et de w dans le tableau. La complexité devra être de l'ordre de la hauteur de l'arbre le plus profond.
  - Montrer que chaque sommet qui n'est pas une feuille ni la racine a degré entrant au moins 2.
  - En déduire que, dans le pire des cas, la profondeur de l'arbre est $\Omega(\log n)$ à la fin de l'algorithme.
  - En déduire une amélioration du temps d'exécution de l'algorithme de Kruskal.

2.3 Algorithme de Prim (non traité en 2022)

On considère maintenant un autre algorithme, dit Algorithme de Prim (Algorithme 5

Procédure 5 Algorithme de Prim

Input: Un graphe pondéré G , un sommet v .

Output: Un arbre couvrant de G de poids minimum.

$F = \emptyset$

$X = \{v\}$

while $X \neq V$ **do**

  Soit uv l'arête de poids minimum entre $u \in X$ et $v \in V \setminus X$

  Rajouter uv à F

  Rajouter v à X

end while

Renvoyer F

Theorem 9. L'algorithme de Prim renvoie un arbre couvrant minimum.

Proof. La preuve est similaire à celle de Kruskal. Il faut simplement bien justifier que l'arête f peut être rajoutée quand e l'est. □

2.4 Nombre d'arbres couvrants

Lemma 10. Une clique admet $n!/2$ chemins couvrants.

Proof. La preuve se fait par induction sur n . On montre que si le premier sommet est fixé alors on a $(n - 1)!$ chemins qui commencent en ce point. En effet on choisit parmi ses $n - 1$ voisins le sommet suivant dans le chemin et par récurrence on a $(n - 2)!$ chemins qui commencent sur le voisin choisi.

Comme on a n choix de sommets initiaux, ça donne $n!$. Mais on "double compte" certains chemins car un chemin peut se lire "dans les deux sens". Chaque chemin est donc énuméré deux fois, ce qui donne la conclusion. □

Quel est donc le nombre d'arbres couvrants d'une clique? On peut montrer que ce nombre est bien plus grand (la preuve initiale est de Cayley):

Lemma 11. Une clique a n^{n-2} arbres couvrants.

Exercice 12. Quel est le nombre de chemins couvrants de:

- Un cycle C_n à n .
- Une roue W_n à $n + 1$ sommets qui consiste en un sommet universel (connecté à tous les autres) plus un cycle à n sommets.

2.5 Exercices

Exercice 13. Donner un algorithme pour trouver un arbre couvrant de poids maximum d'un graphe.

3 Algorithmes gloutons

3.1 Définitions et exemples

Informellement un algorithme glouton est un algorithme pour lequel un choix optimal local va conduire à un optimum global. Autrement dit c'est un algorithme du type de l'algorithme 6.

Procédure 6 Glouton

Input: Une instance I d'un problème Π .

Output: Une solution de I .

$S = \emptyset$

$I' = I$

while I' n'est pas trivial **do**

    Rajouter le "meilleur" élément x de I' dans S .

    Simplifier I' en conséquence

end while

Renvoyer S

Remarquons que l'algorithme de Kruskal est un algorithme glouton puisqu'à chaque étape l'algorithme consiste à compléter la solution actuelle avec l'arête de poids minimum qui maintient une solution. De même l'algorithme de Prim est aussi un algorithme glouton puisqu'à chaque étape on étend la solution de la meilleure façon possible "par rapport à une certaine règle".

L'avantage des algorithmes glouton est leur efficacité temporelle (leur temps d'exécution) et leur simplicité à mettre en oeuvre. Il s'agit en général d'algorithmes quasi-linéaires (ou au pire quadratique) qui sont également très simples à implémenter. Notez néanmoins que pour le temps d'exécution cela dépend du temps nécessaire pour trouver le "meilleur" élément. Comme on l'a vu pour l'algorithme de Kruskal, le temps d'exécution -si on implémente de manière l'algorithme- est quadratique et pas linéaire.

Exemple d'algorithmes gloutons On vient de voir dans les paragraphes précédents des algorithmes gloutons. L'algorithme de Prim et Kruskal sont de tels algorithmes. Mais pour tout problème on peut définir un (ou des) algorithmes gloutons. On peut alors se poser la question de l'efficacité de tels algorithmes... Donnons d'abord un ou deux exemples supplémentaires.

Le problème de la base de poids minimum est le suivant: On se donne m vecteurs de dimension n qui est génératrice de $\mathbb{R}^n$ avec une fonction de poids pour les vecteurs. Le but est de trouver la base de poids minimum.

Theorem 14. *L'algorithme glouton qui consiste à essayer de rajouter l'élément de poids minimum à chaque étape est un algorithme optimal.*

Proof. (Comme pour la preuve de l'algorithme de Kruskal on va supposer que tous les poids sont différents par simplicité).

La preuve est par contradiction. Supposons que la famille renvoyée par l'algorithme est $B = v_1, \dots, v_n$ alors que la famille optimale est $B^* = v_1^*, \dots, v_n^*$. Supposons que ces vecteurs sont classés par ordre croissant de poids. Soit v l'élément de poids maximum qui appartient à $B \setminus B^*$ ou à $B^* \setminus B$. Montrons que si $v \in B^*$ alors B^* n'est pas de poids minimum et si $v \in B$ alors on n'a pas appliqué la règle de notre algorithme glouton.

Si $v \in B^* \setminus B$ alors, par théorème de la base incomplète on peut compléter $B^* \setminus v$ alors un élément de B (qui est une base). Comme v^* est l'élément de poids maximum qui n'est pas dans $B \cap B^*$, l'élément de B que l'on rajoute à un poids inférieur à celui de v , une contradiction avec la minimalité du poids de la base B^* .

Si $v \in B \setminus B^*$ alors il existe un élément $w \in B^*$ qui peut être rajouté à $B \setminus v$ et tel que le poids de w est inférieur à celui de v . Par définition de notre algorithme glouton w est considéré avant v . Soit B_w l'ensemble de vecteur sélectionné au début de l'étape correspondant à w . Notons que B_w ne contient pas v . Comme $(B \setminus v) \cup w$ est une base, w peut être rajouté à B_w et, par définition de l'algorithme glouton, aurait du être ajouté. Une contradiction. $\square$

3.2 Pour la culture: La raison du miracle

Un *matroïde* $\mathcal{M} = (V, \mathcal{I})$ est composé d'un ensemble de sommets V et d'un ensemble de sous ensembles de V dénoté par $\mathcal{I}$ qui satisfait la propriété suivante:

- $\emptyset$ est dans $\mathcal{I}$,
- *Hérédité*: Si $X \in \mathcal{I}$ et $Y \subseteq X$ alors $Y \in \mathcal{I}$,
- *Échange*: Si $A, B \in \mathcal{I}$ et $|A| > |B|$ alors il existe $x \in A \setminus B$ tel que $B \cup \{x\}$ est dans $\mathcal{I}$.

Voilà quelques exemples de matroïdes:

- Familles libres d'un espace vectoriel.
- Forêts d'un graphe.

Theorem 15. Soit Π un problème d'optimisation tel que l'ensemble des solutions forment un matroïde. Alors l'algorithme glouton renvoie une solution optimale pour Π .

Exercice 16. Essayer d'adapter la preuve du Théorème 14 pour prouver le Théorème 15.

4 Performance des algorithmes gloutons - Vertex Cover

Soit $G = (V, E)$ un graphe. Un *vertex cover* (ou couverture par les sommets) de G est un ensemble X de sommets de G tel que pour tout arête $uv \in E$ alors u ou v sont dans X . Les vertex covers sont des objets important en graphes, par exemple pour la surveillance de réseaux.

Procédure 7 Algorithme glouton pour Vertex Cover

Input: $G = (V, E)$ un graphe.

Output: X un vertex cover de G .

$X = \emptyset$

$G' = G$

while G' contient une arête **do**

    Soit x le sommet de degré maximum

    Rajouter x dans X .

$G' \leftarrow G' \setminus x$ (le graphe G' où x et toutes les arêtes qui lui sont incidentes sont supprimées)

end while

Renvoyer X

Lemma 17. L'Algorithme 7 n'est pas forcément optimal pour le problème du vertex cover.

Proof. Considérons par exemple une étoile à trois branches subdivisée une fois. L'algorithme glouton donne une solution de 4 alors que l'optimal renvoie une solution de 3. $\square$

Notons néanmoins que l'algorithme glouton renvoie toujours une solution *minimale par inclusion*. Mais cette solution minimale par inclusion n'est pas nécessairement *minimum*. On peut néanmoins avoir l'impression que la solution ne sera qu'au plus 2 fois celle de la meilleure solution possible (on appellera ça une 2-approximation dans la prochaine section). Mais ce n'est en fait pas le cas:

Theorem 18. L'algorithme glouton peut renvoyer une solution "arbitrairement mauvaise" comparé à la solution optimale.

Proof. La réponse est non. On va créer un graphe biparti avec des parties (A, B) où A contient $k!$ sommets. Notez que dans un graphe biparti, chaque partie est un vertex cover du graphe. En particulier, le graphe que nous allons construire contient un vertex cover de taille au plus $|A| = k!$.

- On crée $(k - 1)!$ sommets B_1 de degré k qui voient chacun k sommets différents de A .
- On crée $k!/(k - 1)$ sommets B_2 de degrés $k - 1$ qui voient chacun $k - 1$ sommets différents dans A .
- ...
- On crée $k!/2$ sommets B_{k-2} de degré 2 qui voient chacun des sommets différents dans A .

Exercice 19. 1. Montrer que les sommets de A ont degré $k - 1$.

2. Montrer qu'après la suppression de $B_1, \dots, B_i$, les sommets de A ont degré $k - i$.

3. En déduire que l'algorithme glouton renvoie $B_1, \dots, B_{k-2}$.

Donc l'algorithme glouton renvoie un ensemble de taille $k!(\frac{1}{k} + \frac{1}{k-1} + \dots + 1) \approx k! \log k$ alors qu'une solution optimale a taille au plus $k!$. $\square$

5 Complexité et algorithme d'approximation

5.1 Enjeux de théorie de la complexité

La théorie de la complexité consiste à *classifier les problèmes selon leur difficulté*. Informellement, certains problèmes ont l'air "plus difficiles" que d'autres. Est-ce parce qu'on n'arrive pas à les résoudre ou existe-t-il vraiment une raison qui fait que ce problème est plus difficile?

La théorie de la complexité tente de répondre à cette question. En particulier, une notion a été introduite dans les années 70: la classe NP. Un problème appartient à NP si "on peut le résoudre en temps polynomial avec un algorithme non déterministe". Sans rentrer dans le détail, cela signifie que si, à chaque étape, un oracle nous donne la meilleure solution alors on peut résoudre le problème et se convaincre que, ce que l'on obtient, est bien une solution à notre problème.

Une autre façon de définir NP est sans doute plus intuitive: si un oracle nous donne une solution à un problème Π alors on peut vérifier qu'il s'agit bien d'une solution à notre problème en temps polynomial. Autrement dit un problème est dans NP si "il admet un certificat polynomial".

Parmi les problèmes de NP, certains sont plus difficiles que d'autres. Les plus difficiles sont les problèmes NP-complets. On conjecture que les problèmes NP-complets ne peuvent pas être décidés en temps polynomial. Autrement dit "il n'existe pas d'algorithmes polynomiaux pour décider un problème NP-complet".

Le problème vertex cover en fait partie. Autrement dit:

Theorem 20. *Le problème VERTEX COVER est NP-complet.*

5.2 Résoudre des problèmes NP-complets - Algorithmes d'approximation et heuristiques

Il est donc illusoire de vouloir résoudre efficacement des problèmes NP-complets. Paradoxalement, la plupart des problèmes à résoudre sont des problèmes NP-complets. Donc il faut quand même trouver un façon de les résoudre... Plusieurs approches sont possibles:

- Restreindre les instances. Certains problèmes sont NP-complet, mais on peut trouver des algorithmes efficaces pour les résoudre dans le cas où l'instance a certaines propriétés. Or dans la plupart des applications, les graphes sont structurés (structure planaire, de grille, de degré maximum borné...etc...). On peut donc essayer de se baser sur ces propriétés supplémentaires pour résoudre efficacement les problèmes.

- Heuristiques : Une heuristique est un algorithme “sans garantie de performance” qui “fonctionne bien en pratique”. Au final les heuristiques se basent un peu sur le point précédent: les instances ne sont pas quelconques et donc on peut trouver des algorithmes -dont on ne prouve pas l'efficacité- qui va tirer partie de la structure pour renvoyer des solutions “efficaces” en pratique.
- Algorithmes d'approximation. Un algorithme d'approximation est un algorithme pour lequel on va pouvoir prouver que la solution renvoyée n'est “pas très loin” de la solution optimale. Autrement dit, c'est un algorithme pour lequel on peut ****garantir**** que la solution qui sera renvoyée ne sera pas très loin de la solution optimale.

5.3 Algorithmes d'approximation

Un problème d'optimisation est un problème où le but est d'optimiser une certaine quantité. Un problème d'optimisation est un problème de *maximisation* (resp. *minimisation*) si la fonction à optimiser est une fonction de maximisation (resp. minimisation). Par exemple, l'arbre couvrant de poids minimum est un problème de minimisation. Une solution d'un problème de maximisation (resp. minimisation) est *optimale* si, parmi l'ensemble des solutions possibles, elle atteint la valeur maximum (resp. minimum).

Un algorithme d'approximation $\mathcal{A}$ est une *c-approximation* pour un problème de maximisation (resp. minimisation) Π est un problème qui, étant donné une instance I renvoie une solution de taille au plus $c \cdot OPT(I)$ (resp. OPT/c) où OPT est la taille de la solution optimale. Autrement dit on garantit que l'algorithme renvoie une solution de taille optimale à un facteur multiplicatif près.

Assez souvent, un des points cruciaux pour évaluer l'efficacité d'un algorithme d'approximation est d'arriver à se comparer à une solution optimale qui, puisqu'on n'arrive pas à la calculer efficacement, est en général assez difficile à manipuler. Autrement dit la question centrale que l'on doit se poser est souvent la suivante:

Comment peut-on se comparer à une solution optimale que l'on ne connaît pas?

La réponse est assez variable, mais dans la plupart des cas, on ne va pas se comparer à cette solution mais on va se comparer à quelque chose qui est plus petit/grand que la solution et qui va nous donner une garantie sur la taille de la solution optimale. Pour l'instant, c'est certainement très flou... On va illustrer ça dans la section suivante pour le problème du vertex cover minimum avec la notion de couplage.

5.4 Maximum et maximal

Soit I un instance d'un problème d'optimisation. Une solution de I est maximum (ou *minimum*) si la taille de cette solution est maximum (resp. minimum) parmi toutes les solutions.

Une solution de I est maximale (resp. *minimal*) s'il est impossible d'étendre la solution en ajoutant un nouvel élément (resp. tout sous ensemble de la solution n'est pas une solution). Autrement dit maximal / minimal signifie simplement être un “optimum local” et être minimal ou maximal par inclusion sans aucune garantie sur la taille de la solution.

Un algorithme glouton renvoie une solution maximal/minimale. Le but est de savoir quand elle maximum / minimum !

Calculer une solution minimale est en général “simple” tandis que trouver une solution minimum est en général “difficile” !

6 Couplage et dualité

6.1 Problème de couplage

Soit $G = (V, E)$ un graphe. On note n le nombre de sommets de G et m son nombre d'arêtes. Un *couplage* est un ensemble d'arêtes X deux à deux sommets-disjointes dans le graphe (si $e = (u, v)$ et $e' = (x, y)$ sont dans X alors $x \neq u, v$ et $y \neq u, v$). L'objectif est de trouver un couplage de taille maximum dans un graphe G .

La recherche d'un couplage de taille maximale joue un rôle central en informatique (en particulier dans les graphes bipartis). Que ce soit en ordonnancement (affectations de tâches à des processus), en optimisation (affectations qui maximisent le bien-être global) ou en sociologie et économie (étude de la stabilité d'une structure). Par exemple, si on a un centraliseur dont le rôle est de distribuer les tâches aux différents processus, on désire affecter les tâches à des processeurs qui sauront les réaliser efficacement. De plus on veut perdre le minimum de temps, et donc affecter le maximum de tâches à chaque étape.

Donnons une application particulière du problème de couplage qui vous a certainement concerné: l'affectation d'étudiants dans les écoles. L'algorithme "Parcours Sup" -dont on a beaucoup parlé ces derniers mois- est un algorithme basé sur la recherche d'un meilleur couplage dans un graphe. Son prédécesseur, APB, était lui aussi basé sur un algorithme similaire. Dans ce cas, on a deux types de sommets: les sommets "écoles" et les sommets "étudiants". Chaque école propose un ensemble d'étudiants qui l'intéressent (en fait, dans Parcours Sup, chaque école classe les étudiants). Et chaque étudiant propose un ensemble d'écoles (dans un monde parfait, il faudrait aussi que chaque étudiant classe les écoles, l'algorithme convergerait plus vite et les gens sauraient avec le 10 Septembre où il se retrouve... Mais c'est un autre problème). Notons A les écoles et B les étudiants. On crée une arête (a, b) entre $a \in A$ et $b \in B$ si et seulement si a est intéressé par b et b est intéressé par a . Pour chaque école a dans A , on a une capacité maximum c_a . Le but est trouver un maximum de paires (a, b) deux à deux disjointes telles que (i) chaque étudiant b est associé à au plus une école; (ii) chaque école a est dans au plus c_a couples.

6.2 Algorithme d'approximation glouton

Le but de cette partie est de prouver le résultat suivant:

Procédure 8 Algorithme glouton pour Vertex Cover

Input: $G = (V, E)$ un graphe.

Output: M un couplage de G .

$M = \emptyset$

$G' = G$

while G' contient une arête e **do**

    Rajouter $e = uv$ dans M .

$G' \leftarrow G' \setminus \{u, v\}$ (le graphe G' où les deux extrémités de e sont supprimées et toutes les arêtes qui lui sont incidentes sont supprimées)

end while

Renvoyer M

Lemma 21. *L'Algorithme glouton 8 donne une 2-approximation du problème de couplage maximum.*

Proof. Soit $e_1, \dots, e_k$ les arêtes de la solution renvoyée par l'algorithme glouton et u_i, v_i les deux extrémités de e_i . Supposons par contradiction qu'il existe un couplage C^* de taille au moins $2k + 1$. Comme les arêtes d'un couplage sont deux à deux disjointes, il existe une arête e de C^* qui n'intersecte pas $\cup_{i \leq k} \{u_i, v_i\}$. Or cette arête n'a pas été rajouté par l'algorithme glouton quand il l'a considéré, une contradiction. $\square$

Mais, de manière peut être un peu magique à première vue, on a aussi le résultat suivant qui relie vertex cover et couplage maximum:

Lemma 22. *La taille maximum d'un couplage est au plus la taille minimum d'un vertex cover.*

Proof. Montrons que, pour tout couplage de taille m , on a un vertex cover de taille au moins m . En effet, chaque arête du couplage doit être touché par un sommet du vertex cover. Donc un vertex cover de taille minimum a au moins la taille d'un couplage maximum. $\square$

Lemma 23. *Vertex Cover admet un 2-approximation.*

Proof. La preuve est très simple. Il suffit de calculer gloutonnement un couplage $e_1, \dots, e_k$. Soit $u_i v_i$ l'arête e_i . Le graphe $G[V \setminus \{\cup_{i \leq k} \{u_i, v_i\}\}]$ est un graphe sans arête. En effet, s'il existait une arête dont les deux extrémités étaient dans $V \setminus \{\cup_{i \leq k} \{u_i, v_i\}\}$ alors elle aurait pu être ajoutée au couplage, en contradiction avec le fait que le couplage était maximal par inclusion. Donc $\cup_{i \leq k} \{u_i, v_i\}$ est un vertex cover.

Prouvons maintenant que n'importe quel vertex cover a taille au moins k . Cela est dû au fait que les arêtes e_i sont extrémités disjointes. Donc s'il existait un vertex cover X de taille au plus $k - 1$, il existerait une arête e_i avec $i \leq k$ telle que ni u_i ni v_i sont dans X . Une contradiction avec la définition de vertex cover. $\square$

La raison de la magie... En fait ce n'est pas surprenant, les deux problèmes sont duaux l'un de l'autre pour la programmation linéaire. Quoi que ça signifie exactement, la programmation linéaire donne "naturellement" un nouveau problème dont la taille d'une solution optimale donnera une borne inférieure pour le problème initial. Il s'agit d'un outil central en informatique et en optimisation que nous n'aurons malheureusement pas le temps de traiter dans ce cours...

6.3 Couplage dans les arbres

On a vu tout à l'heure que l'algorithme glouton pour les graphes en général ne donnait pas forcément une solution optimale. Essayons de l'adapter en étant plus malin et en "choisissant" mieux l'arête e . On se propose d'utiliser la règle suivante:

On rajoute une arête incidente à un sommet de degré minimum dans le graphe restant.

Exercice 24. Cet algorithme est optimal dans le cas des arbres. Autrement dit si G est un arbre algorithme cet algorithme renvoie un couplage de taille maximum.

6.4 Couplage dans les graphes bipartis

La *différence symétrique* de deux ensembles A, B notée $\Delta(A, B)$ est $A \setminus B \cup B \setminus A$.

Soit M_1, M_2 deux couplages:

Remark 25. $\Delta(M_1, M_2)$ induit une collection de chemins et de cycles pairs.

Étant donné un couplage M_1 , un sommet est *libre* s'il n'intersecte aucune arête de M_1 . Un *chemin alternant* est un chemin $v_1, \dots, v_i$ tel que v_1 est libre et le chemin $v_1, \dots, v_i$ est une alternance d'arêtes pas dans M_1 et d'arêtes dans M_1 .

Un chemin alternant est dit *augmentant* s'il commence et termine par des sommets libres.

Lemma 26. *S'il existe un chemin augmentant pour M_1 alors M_1 n'est pas maximum.*

Proof. Il suffit d'inverser sur le chemin les arêtes et les non arêtes pour augmenter la taille du couplage. On remarque qu'il s'agit bien d'un couplage puisque les premiers et derniers sommets du chemin augmentant sont libres. $\square$

Lemma 27. *Dans un graphe biparti, il est possible de trouver un chemin augmentant en temps polynomial (si un tel chemin augmentant existe).*

Theorem 28 (Berge). *Un couplage est maximum si et seulement si il n'a pas de chemin augmentant.*

6.5 Et en général?

Theorem 29 (Edmonds). *Le problème du couplage maximum admet un algorithme polynomial dans les graphes.*

7 Indépendant Maximum

7.1 Algorithme glouton

Un *ensemble indépendant* est un sous ensemble de sommets X du graphe tel que il n'existe aucune arête entre les sommets de X . Une *clique* est un ensemble de sommets deux à deux reliés dans un graphe.

Exercice 30. Montrer que s'il existe un algorithme $\mathcal{A}$ permettant de calculer un indépendant de taille maximum (pour tous les graphes) alors il existe un algorithme qui permet de calculer une clique de taille maximum.

Indication: Remplacer les arêtes par les non-arêtes.

On propose l'algorithme 9 pour résoudre de problème de l'indépendant maximum dans un graphe.

Procédure 9 Algorithme glouton pour Ensemble Indépendant

Input: $G = (V, E)$ un graphe.

Output: X un ensemble indépendant de G .

$X = \emptyset$

$G' = G$

while G' contient un sommet v **do**

    Rajouter v dans M .

$G' \leftarrow G' \setminus N(v)$ (le graphe G' où les voisins de v (et v) sont supprimés ainsi que toutes les arêtes qui lui sont incidentes sont supprimées)

end while

Renvoyer X

Exercice 31. 1. Donner un graphe pour lequel l'Algorithme 9 n'est pas optimal.

2. Quelle (in)efficacité arrivez vous à atteindre pour cet algorithme en général?

3. Soit I la solution renvoyé par l'algorithme 9. Montrer que I contient au moins $n/(\Delta + 1)$ sommets, où Δ est le degré maximum du graphe.

Indication: Comment de sommets sont supprimés à chaque étape?

4. En déduire que l'algorithme 9 est une $(\Delta + 1)$ -approximation pour le problème de l'indépendant maximum.

Indication: Utiliser une borne supérieure (triviale) sur la taille d'une solution !

5. Montrer que si un sommet $v \in I$ n'est pas dans I^* alors, quitte à supprimer au plus Δ sommets de I^* , on peut le rajouter à I^* .

6. En déduire que l'algorithme 9 est en fait une Δ -approximation.

7.2 Problèmes d'arêtes et line graphs

Le *line graph* d'un graphe $G = (V, E)$ est le $L(G) = (E, E')$ où les sommets de $L(G)$ sont les arêtes de G et on relie deux sommets e_1, e_2 de $L(G)$ si les arêtes correspondantes dans G partagent un sommet.

Pour jouer un peu avec cette notion, regardons les exemple suivants:

Exercice 32. Quel est le line graph d'une étoile à n feuilles? Un chemin à n sommets?

Exercice 33. Trouver le lien entre ensembles indépendants de G et couplages de $L(G)$.

Plus globalement, il existe de nombreux problèmes qui sont les "versions arêtes" d'autres problèmes en "version sommet". Ces problèmes sont plus simples que leur contre-parties sommets car cela signifie qu'ils correspondent au problème "sommet" sur des instances particulière qui correspondent à des "graphes d'arêtes". Essayons de formaliser ça en quelques mots.

Les line graphs ont de nombreuses propriétés qui les rendent plus simples que les graphes généraux. Par exemple:

- Exercice 34.**
1. Montrer que le degré maximum de $L(G)$ est au plus deux fois le degré maximum de G .
  2. Montrer que le voisinage d'un sommet dans un line graph est une union d'au plus deux cliques. Autrement dit, pour tout e de $L(G)$, $N(e)$ peut être partitionné en deux parties X_1, X_2 tels que X_1 et X_2 forment des cliques.
Indication: Supposez par l'absurde qu'il existe dans $N(e)$ trois sommets e_1, e_2, e_3 deux à deux non adjacents.
  3. En déduire que l'algorithme glouton de l'indépendant maximum dans les line graphs donne une 2-approximation.
Indication: (difficile) Adapter l'argument de l'exercice 31 en remplaçant Δ par le nombre de cliques dans le voisinage.
  4. Comparer cet algorithme à l'algorithme glouton que nous avons utilisé précédemment.

8 Domination