Algorithmique Avancée
Partie 1 - Graphes

Nicolas Bousquet

September 27, 2021

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

- Algorithmes gloutons.
- Algorithmes d'approximation.
- Algorithmes dynamiques.

1 Definitions et notations

1.1 Definition usuelles

Un *graphe* $G = (V, E)$ est composé d'un ensemble V de *sommets* et d'un ensemble E de paires d'éléments de V appelées *arêtes*. En d'autres termes un graphe est un schématiquement résumé en 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 $f : E \to \mathbb{R}$ 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. $n = |V(G)|$ et m le nombre d'arêtes de G.

Soit v un sommet de 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 $N(v)$.

Un *chemin* est une suite $v_1, v_2, \ldots, v_{k+1}$ de sommets sans répétition tel que pour tout $i \leq k$ v_iv_{i+1} est une arête. Si, en plus $v_{k+1}v_1$ est une arête alors $v_1, v_2, \ldots, v_{k+1}$ est un *cycle*. La *longueur* d'un chemin est son nombre de sommets moins 1. Autrement dit la longueur du chemin $v_1, v_2, \ldots, v_{k+1}$ 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. 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) tous les plus petits sous ensemble Y de X ont une arêtes entre Y et $V \setminus Y$.

Exercice 1. Montrer que $\sum_{v \in V} d(v)$ est pair.

1.2 Quelques familles de graphes importantes

Une grille de taille $n \times m$ est un graphe tel que les sommets sont indexées 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, denotée 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 denoté par $K_{n,m}$.

Un *arbre* est un graphe connexe sans cycle.

Exercice 2. 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.

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 forcement 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-eme sommet v_i. Autrement dit la i-eme 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 strutures 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).

Inconvenients: Espace quadratique.

2 Arbre couvrant

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 sommet 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 transferer 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

Etant 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 est dit de *profondeur* i si l'unique chemin de v vers w a longueur i. On dit par convention que la racine a pronfondeur 0.

Exercice 3.
- Montrer que l'arc ab ne peut pas être orienté à la fois de a vers b et de b vers a.

Un parcours en largeur de G en partant de v est un arbre construit grace 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.

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 viendra le second arrivé...etc...

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 = ∅
F = ∅
while L ≠ ∅ 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 = ∅
F = ∅
while L ≠ ∅ 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
```

- 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-même être effectuées.
  Notez que les listes en informatique sont en général des structures FILO !

Exercice 4.
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. Montrer que si on fait un BFS alors, si u est au i-eme niveau, alors $d(u, v) = i$.
4. Soit u au i-eme niveau d'un BFS et w au j-eme niveau du même BFS. Que dire de $d(u, w)$?

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 5.

Existe-t-il un lien entre le niveau maximum d'un BFS / DFS et les diamètres ou plus long chemins dans les graphes?

De manière surprenante, on peut montrer que si on fait du BFS (un peu spéciaux) consécutif alors on approxime à un facteur additionnel 1 près le diamètre d'un graphe !

Exercice 6. Montrer que si G est connexe, l'algorithme renvoie un arbre couvrant. Que renvoie-t-il s'il n'est pas connexe?

Conclusion

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

2.2 Algorithme de Kruskal

Procedure 3 Algorithme de Kruskal

Input: Un graphe pondéré G.
Output: Un arbre couvrant de G de poids minimum.

```
F = ∅
E' = E
while E' n'est pas vide do
    Soit uv l'arête de poids minimum dans E'
    if F ∪ {uv} est acyclique then
        Rajouter uv à F
    end if
    Supprimer uv de E'
end while
Renvoyer F
```

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

Proof. Par induction sur la taille de la différence symmétrique. Si elle est nulle alors la conclusion est directe. Sinon considérons l'arête e de poids maximum renvoyé par l'arbre couvrant non optimal qui n'est pas dans l'arbre couvrant optimal. Soit T^* l'arbre optimal et T l'arbre renvoyé.

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. □

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 X. Pour faire ça, on va tenir un tableau de taille n qui, à chaque sommet lui associe le numéro de sa composante connexe.

Initiallement, les composantes connexes sont numérotées de 1 à n et chacun des sommet v_i est numéroté i. Ensuite, on considère l'arête u, v dans l'algorithme:

- Si les numéros de la composante de u et de v sont les mêmes, on n'ajoute pas l'arête à F, sinon on l'ajoute.
- Si les deux composantes ont des numéros différent, on ajoute l'arête. On fusionne alors les composantes de u et v. Autrement dit, on prend tous les éléments qui ont comme valeur le numéro de la composante de u et on leur donne le numéro de la composante de v.

Quelle est la complexité de l'algorithme?

- 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.

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 representer les differentes composantes connexes via un arbre plutot 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'ancetre de v dans S si v n'est pas la racine de sa composante et son label est v sinon.

Exercice 8.
1. (Find) Montrer que l'on peut verifier 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.
2. (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.
3. Montrer que, dans le pire des cas, la profondeur de l'arbre est $O(\log n)$ à la fin de l'algorithme.
4. En déduire que l'algorithme de Kruskal peut s'exécuter en temps $O(n \log n)$.

2.3 Algorithme de Prim

On considère maintenant un autre algorithme, dit Algorithme de Prim (Algorithme 4

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. □

Procedure 4 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

2.4 Nombre d'arbres couvrants

Lemma 10. *Une clique admit $n!/2$ 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 recurrent 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?

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

Exercice 12.
- Prouver cette fonction pour $n = 4$.
- Donner la formule de recurrence satisfaite pour des arbres de taille n.

2.5 Exercices

Exercice 13. Soit $G = (V, E)$ un graphe, s un sommet et k un entier. Proposer un algorithme qui trouve la sous graphe acyclique connexe à au plus k arêtes qui contient s et a poids minimum.

Hint: Notez que si $k = |V|$, le problème est équivalent à chercher un arbre couvrant de poids minimum pour G. Proposez une variante de l'algorithme de Prim pour prouver ce résultat.

- Cet algorithme fournit-il une forêt avec une composante de taille k?
- Si oui, est-il minimum?

Exercice 14. 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 5.

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".

Procedure 5 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" element x de I' dans S.
  Simplifier I' en conséquence
end while
Renvoyer S

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 de poser la question de l'efficacité de tels algorithmes... Donnons d'abord un ou duex 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 15. *L'algorithme glouton est un algorithme optimal.*

Proof. La preuve est par contradiction. Supposons que la famille renvoyée par l'algorithme est $B = v_1, \ldots, v_n$ alors que la famille optimale est $B^* = v_1^*, \ldots, v_n^*$. Supposons que ces vecteurs sont classées par ordre croissant de poids.

Soit v_i^* l'element de B^* de poids minimum qui est dans $B^* \setminus B$. TODO □

Remarquez que dans la preuve précédente, on aurait du montrer que la famille renvoyée par l'algorithme est génératrice. Exercice: faites le !

3.2 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 $Y \in \mathcal{I}$,
- *Echange*: 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 matroides:

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

Theorem 16. *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.*

4 Quelle efficacité pour un algorithme glouton?

Autant dans le cas des graphes pondérés on voit ce qu'est un algorithme glouton (on étend la solution partielle) en une nouvelle solution avec une nouvelle arête / un nouveau sommet de coût minimum. Mais la définition de ceci est moins claire quand on n'est pas pondéré. Un algorithme glouton fait "le meilleur choix possible" en fonction d'une règle simple.

4.1 Vertex Cover

Soit $G = (V, E)$ un graph. 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 centraux en graphes, par exemple pour la surveillance de réseaux.

Procedure 6 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 6 n'est pas forcement optimal pour le problème du vertex cover.*

Proof. Considerons 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. □

Notons néanmoins que l'algorithme glouton renvoie toujours une solution *minimale par inclusion*. Mais cette solution minimale par inclusion n'est pas formement *minimum*.

Exercice 18. Est-ce que l'algorithme glouton pour Vertex Cover donne une solution de taille au plus deux fois la taille de la solution optimale?

La réponse est non. Schématiquement l'exemple est le suivant:

- On aura un graphe biparti où la premiere partie A contient $k!$ sommets. En particulier cette partie est un VC.
- On crée $(k-1)!$ sommets de degré k qui voient chacun k sommets différents de A.
- On crée $k!/(k-1)$ sommets de degrés $k-1$ qui voient chacun $k-1$ sommets différents dans A.
- ...
- On crée $k!$ sommets de degré 1 qui voient chacun des sommets différents dans A.

On note que l'algorithme glouton peut renvoyer tous les sommets $V \setminus A$ puisqu'initialement les sommets de degré k ont le même degré que les sommets de A. Quand ils sont supprimés, les sommets de A ont degré $k-1$ et donc on peut selectionner les sommets de $V \setminus A$ de degré $k-1$...etc...

Donc au total l'algorithme glouton peut renvoyer un ensemble de taille $k!(\frac{1}{k} + \frac{1}{k-1} + \ldots + 1) \approx k! \log k$.

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, celà 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 intuiive: 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 19. *Le probleme* 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 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 point 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.

On peut se poser la question de ce qu'est une "bonne" borne inférieure pour la taille d'un couplage minimum. Et la réponse va se trouver dans la section suivante: il s'agit de 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 taile 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 élement (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 un solution maximal/miinimale. Le but est de savoir quand elle maximum / minimum !

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* 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 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'affectaion 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 classes 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:

Procedure 7 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 X

Lemma 20. *L'Algorithme glouton 7 donne une 2-approximation du problème de couplage maximum.*

Proof. Soit $e_1, \ldots, e_k$ 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 extrémités 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. □

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

Proof. Montrons que, pour tout matching 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. □

Lemma 22. *Vertex Cover admet un 2-approximation.*

Proof. La preuve est très simple. Il suffit de calculer gloutonnement un couplage $e_1, \ldots, e_k$. 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 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. Celà est du 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. □

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 forcement 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 23. 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 symmetrique 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 24. $\Delta(M_1, M_2)$ induit une collection de chemins et de cycles pairs.

Etant donné un couplage M_1, un sommet est *libre* s'il n'intersecte aucune arête de M_1. Un *chemin alternant* est un chemin $v_1, \ldots, v_i$ tel que v_1 est libre et le chemin $v_1, \ldots, v_i$ 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 25. *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 tailel du couplage. On remarque qu'il s'agit bien d'un couplage puisque les premiers et derniers sommets du chemin augmentant sont libres. □

Lemma 26. *Dans un graphe biparti, il est possible de trouver un chemin augmentant en temps polynomial (si un tel chemin augmentant existe).*

Theorem 27 (Berge)**.** *Un couplage est maximum si et seulement si il n'a pas de chemin augmentant.*

6.5 Et en général?

Theorem 28 (Edmonds)**.** *Le problème du couplage maximum admet un algorithme polynomial dans les graphes.*

7 Independent Maximum

7.1 Algorithme glouton

Un *ensemble indépendent* 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 29. 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.

TODO - Algorithme glouton.

Exercice 30.
1. Donner un graphe pour lequel l'algorithme glouton n'est pas optimal.
2. Quelle (in)efficacité arrivez vous à atteindre pour cet algorithme en général?

On peut néanmoins donner une garantie pour cet algorithme:

Lemma 31. *Soit G un graphe. L'algorithme glouton pour l'indépendant maximum renvoie un ensemble indépendant de taille au moins $n/(\Delta+1)$ où Δ est le degré maximum du graphe.*

En particulier, l'algorithme est une $\Delta+1$ approximation pour le problème du calcul de l'ensemble maximum de l'ensemble indépendant.

Proof. Pour le montrer, montrons que tout indépendant maximal par inclusion est a taille au moins $n/(\Delta+1)$. Par contradiction, supposons qu'un ensemble X de taille inférieure stricte à $n/(\Delta+1)$ soit maximal par inclusion. Alors $X \cup N(X)$ couvre les sommets du graphe. Cet ensemble a taille au plus $|X| + \Delta \cdot |X|$ qui est, par hypothèse, strictement inférieur à n, une contradiction. □

7.2 Problèmes d'arêtes et line graphs

Exercice 32. Trouver le lien entre ensembles indépendants et couplages.

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 pendant sommets car celà 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.

Un graphe $G = (V, E)$ est un *line graph* si il existe $H = (W, V)$ tel que les sommets de G correspondent aux arêtes de H et uv est une arête de G si et seulement si les arêtes u et v de H partagent une extrémité commune. Les line graphs ont de nombreuses propriétés qui les rendent plus simples que les graphes généraux. Par exemple:

Lemma 33. *Soit $G = (V, E)$ un line graph. Pour tout sommet $v \in V$, $N(v)$ peut être partitionné en deux cliques.*