Algorithmique des graphes – quelques notes de cours

Ioan Todinca, avec le concours de Julien Tesson

29 avril 2008

Table des matières

Première partie

Algorithmes de base

Chapitre 1

Généralités

1.1 Définitions et notations

Définition 1 (Graphe non orienté) *Un graphe* non-orienté *est un couple $G = (X, A)$ où X est l'ensemble des* sommets *et A est l'ensemble d'*arêtes *de G. Chaque arête est une paire de sommets. On notera xy l'arête $\{x, y\}$.*

Définition 2 (Graphe orienté) *Un graphe* orienté *est un couple $G = (X, U)$ où X est l'ensemble des* sommets *et A est l'ensemble d'arcs de G. Chaque arête est un couple de sommets. On notera (xy) l'arête (x, y).*

Remarque 3 *Dans un graphe non orienté $G = (X, A)$, l'ordre des extrémités des arêtes ne compte pas, l'arête xy est identique à l'arête yx. Dans un graphe orienté $G = (X, U)$, les arcs (xy) et (yx) n'ont pas la même signification.*

Définition 4 (adjacence, incidence, voisinage, degré) *Deux sommets x et y de $G = (X, A)$ sont* adjacents *si $xy \in A$.*

Deux arêtes dont incidentes *si elles partagent une même extrémité. L'arête xy est incidente aux sommets x et y.*

Le voisinage *d'un sommet x est $\Gamma(x) = \{y \in X \mid xy \in E\}$. On note $d(x)$ de* degré *de x, i.e. le nombre de voisins de x.*

Définition 5 (chaîne, cycle) *Soit $G = (X, A)$ un graphe non orienté. On appelle* chaîne *de G une suite se sommets $\mu = [x_1, x_2, \ldots, x_p]$ telle que $x_i x_{i+1} \in A$, $\forall i, 1 \leq i \leq p-1$. Un* cycle *est une chaîne dont le premier et dernier sommets sont adjacents. Dans les graphes orientés, une chaîne (resp. cycle) qui respecte l'orientation des arcs est appelée* chemin *(resp.* circuit*).*

1.2 Structures de données pour la représentation des graphes

1.2.1 Matrice d'adjacence

1.2.2 Tableau de listes des successeurs

Chapitre 2

Algorithmes de parcours

Les parcours en largeur et en profondeur des graphes généralisent les parcours similaires dans les arbres. Ces algorithmes servent à rechercher des chemins et des cycles dans un graphe, à déterminer les composantes connexes, etc. Ils nous serviront souvent en tant que procédures de base pour d'autres algorithmes.

Lors d'un parcours, un sommet a trois états possibles :

- `non_atteint` – l'algorithme n'a pas encore rencontré ce sommet ;
- `atteint` – le sommet a été rencontré, mais nous n'en avons pas fini avec lui ;
- `traité` – le sommet a été traité, nous avons parcouru toutes les arêtes incidentes à ce sommet (ou les arcs sortants, s'il s'agit d'un graphe orienté).

En plus de son état, chaque sommet x recevra un numéro $\sigma[x]$, qui correspond à l'ordre du parcours. Enfin, pour chaque sommet x on garde le $\mathsf{pere}[x]$, à savoir le sommet duquel nous sommes venus lorsque nous avons atteint x pour la première fois. Cette arborescence pourra s'avérer bien utile par la suite !

2.1 Parcours en largeur

2.1.1 L'algorithme

Lorsque l'on fait un parcours en largeur à partir d'un sommet x, on atteint d'abord les voisins de x, ensuite les voisins des voisins (sauf ceux qui sont déjà atteints) et ainsi de suite. C'est pourquoi le parcours en largeur est aussi appelé *parcours concentrique*. Il sert notamment à le recherche des plus courts chemins dans un graphe. Une description détaillée du parcours en largeur est donnée dans l'algorithme 1.

Algorithme 1 : Parcours en Largeur

```
Entrées : Un graphe G = (X, A)
Sorties : Un ordre σ sur les sommets et une arborescence
//init
pour chaque x ∈ X faire
    etat[x] ← non_atteint
index ← 1
a_Traiter ← ∅
//boucle principale
pour chaque z ∈ X faire
    si etat[z] = non_atteint alors
        //on lance un parcours à partir de z
        pere[z] ← nil
        σ[z] ← index
        index++
        a_traiter.AjouterEnQueue(z)
        tant que a_traiter ≠ ∅ faire
            x ← Tete(a_taiter)
            pour chaque y ∈ Γ(x) faire
                si etat[y] = non_atteint alors
                    etat[y] ← atteint
                    pere[y] ← x
                    σ[y] ← index
                    index++
                    a_traiter.AjouterEnQueue(y)
            a_traiter.EnleverTete()
            etat[x] ← traite
```

2.1.2 Complexité

Si le graphe est donné par tableau de listes de successeurs, la complexité du parcours en largeur est $\mathcal{O}(n+m)$.

2.1.3 Exercices

1. Modifier l'algorithme de parcours en largeur afin de récupérer les composantes connexes du graphe en entrée.

2. Appliquer le parcours en largeur à la recherche d'un plus court chemin entre deux sommets x et y du graphe G.

3. Proposer une version du parcours en largeur où la file a_traiter est simulée à l'aide d'un tableau de n éléments. Il suffira de garder deux variables, *deb* et fin, qui pointeront sur le premier et respectivement le dernier élément à traiter.

2.2 Parcours en profondeur

2.2.1 L'algorithme

Contrairement au parcours en largeur, lorsque l'on fait un parcours en profondeur à partir d'un sommet x on tente d'avancer le plus loin possible dans le graphe, et ce n'est que lorsque toutes les possibilités de progression sont bloquées que l'on revient (étape de backtrack) pour explorer un nouveau chemin ou une nouvelle chaîne. Le parcours en profondeur correspond aussi à l'exploration d'un labyrinthe. L'algorithme 2 propose une implantation récursive du parcours en profondeur.

Les applications de ce parcours sont peut-être moins évidentes que pour le parcours en largeur, mais le parcours en profondeur permet de résoudre efficacement des problèmes plus difficiles comme la recherche de composantes fortement connexes dans un graphe orienté, le test de planarité, etc.

Procédure : Parcours_Prof_Recursif(sommet x)

$etat[x] \leftarrow atteint$;
$\sigma[x] \leftarrow index$;
$index++$;
pour chaque $y \in \Gamma(x)$ **faire**
  si $etat[y] = non\_atteint$ **alors**
    $pere[y] \leftarrow x$;
    Parcours_Prof_Recursif(y);
$etat[x] \leftarrow traite$;

Algorithme 2 : Parcour en Profondeur

Entrées : Un graphe $G = (X, A)$
Sorties : Un ordre σ sur les sommets et une arborescence
//init ; **pour chaque** $x \in X$ **faire**
  $etat[x] \leftarrow non\_atteint$
$index \leftarrow 1$
//boucle principale **pour chaque** $x \in X$ **faire**
  si $etat[x] = non\_atteint$ **alors**
    //on lance un parcours à partir de x
    $pere[x] \leftarrow nil$
    Parcours_Prof_Recursif(x)

2.2.2 Complexité

Comme pour le parcours en largeur, si le graphe est donné par tableau de listes de successeurs, la complexité du parcours en profondeur est $\mathcal{O}(n + m)$.

2.2.3 Exercices

1. Modifier l'algorithme de parcours en profondeur afin de récupérer les composantes connexes du graphe.

2. Appliquer le parcours en profondeur à la recherche d'un chemin entre deux sommets x et y.
3. Utiliser le parcours en profondeur pour chercher un cycle dans un graphe non-orienté.
4. Proposer une implantation non récursive du parcours en profondeur. On pourra utiliser une structure très semblable à l'algorithme de parcours en largeur, où la file `a_traiter` sera remplacée par une pile.

Chapitre 3

Graphes orientés sans circuits. Tri topologique

3.1 L'algorithme

Les *graphes orientés sans circuits* (souvent appelés DAG, comme *directed acyclic graphs*) apparaîssent souvent dans les modélisation des relations ce précédence. L'escemple typique est l'exécution d'un projet composé de tâches, pour lequel on a des informations de type "la tâche A doit être effectuée avant la tâche F".

Théorème 6 *Soit $G(X,U)$ un graphe orienté sans circuit. G possède un* tri topologique*, c'est à dire un ordre total $(x_1, x_2, \ldots, x_n)$ sur les sommets tel que tous les arcs de G sont orientés dans le sens croissant de l'ordre : pour tout arc $(x_i x_j) \in U$, on a $i < j$.*

Preuve. (Idées) Montrer d'abord qu'il existe un sommet x de degré entrant 0 (sans prédécesseurs). Pour ce faire, considérer par exemple un plus long chemin dans le graphe et montrer que le premier sommet n'a pas de prédécesseurs. Ensuite, par récurrence, montrer l'ordre total sur les sommets qui commence par x et continue par un tri topologique de G_x satisfait la condition. ◇

Algorithme 3 : Tri toplogique

Entrées : $G(X,U)$ orienté sans circuit
Sorties : Ordre total σ sur les sommets tel que $\forall\ (xy) \in U$, $\sigma(x)\ < \sigma(y)$
//init
degrés_entrants(G)
$dispo \leftarrow \emptyset$
$\forall x$ *si* $d^-[x] = 0$ *alors* $dispo \leftarrow dispo \cup \{x\}$
pour *i de 1 à n* **faire**
  $x \leftarrow dispo.Tete()$
  $dispo.EnleverTete()$
  $\sigma[x] \leftarrow i$
  pour chaque $y \in \Gamma^+(x)$ **faire**
    $d^-[y] \leftarrow d^-[y] - 1$
    si $d^-[y] = 0$ **alors**
      $dispo.Ajouter(y)$
retourner σ

Procédure :degrés_entrants
Entrées : $G(X,U)$
Sorties : tableau d^- des degrés de chaques sommets
//init;
pour chaque $x \in X$ **faire**
  $d^-[x] \leftarrow 0$;
pour chaque $x \in X$ **faire**
  pour chaque $y \in \Gamma^+(x)$ **faire**
    $d^-[y] + +$;
retourner d^-;

L'algorithme 3 calcule un ordre topologique d'un graphe orienté sans circuit.

3.2 Exercices

1. Montrer que l'algorithme **Tri topologique** est correct.
2. Vérifier que l'algorithme est de complexité linéaire.
3. Modifier l'algorithme pour qu'il prenne en entrée un graphe orienté G quelconque, et que la sortie soit ou bien un tri topologique de G, ou bien le message "G possède un circuit".

Chapitre 4

Plus courts chemins dans un graphe

Considérons maintenant un graphe orienté $G = (X, U)$ tel que chaque arc (xy) ait un coût $w(x, y)$ (on parle également de longueur ou de poids de l'arc). Etant donné un chemin μ dans ce graphe, le coût de μ est la somme des coûts de ses arcs. La *distance* du sommet a au sommet b est le coût minimum parmi tous les chemins de a à b (ou ∞ s'il n'existe pas de tel chemin). Elle sera notée $\sigma(a, b)$. L'objectif de ce chapitre est de donner des algorithmes qui calculent les distances entre les sommets, ainsi que des plus courts chemins les reliant. Dans la suite, seules les distances seront calculées explicitement ; au lecteur de compléter les algorithmes afin de pouvoir récupérer également les plus courts chemins.

On distingue plusieurs types d'algorithmes, selon

- *La nature des coûts* : souvent dans la modélisation tous les coûts sont positifs, c'est pourquoi il existe des algorithmes spécifiques pour ce cas. Ils sont aussi plus efficaces que les algorithmes pour les graphes avec coûts arbitraires.
- *Les chemins recherchés* : parfois on cherche un plus court chemin d'une source s à une cible t données. Pour trouver un tel chemin, on calcule en fait les plus courts chemins de s à tous les autres sommets. Il existe également des algorithmes qui calculent les distances entre chaque paire de sommets.

4.1 Plus courts chemins à partir d'une source, avec coûts positifs

L'algorithme de Dijsktra calcule les plus courts chemins d'un sommet à tous les autres sous l'hypothèse où tous les coûts sont positifs (il n'est pas difficile de trouver un exemple, avec coûts négatifs, où il donne une réponse erronée !). A chaque étape on met dans une liste des sommets traités les sommets dont la distance a été correctement calculée. Les sommets arrivent dans cette liste par distance croissante. Pour les sommets x de la liste A_traiter, le tableau d indique le plus court chemin de la source à x, dont tous les sommets sauf x sont déjà traités.

Algorithme 4 : Dijsktra

Entrées : Un graphe $G = (X, A)$, une fonction $w : U \to \mathbb{R}^+$, un sommet s
Sorties : Un tableau d tel que $d[x] = \sigma(s, x)$ pour tout x
Initpluscourtchemin (G, s)
A_Taiter $\leftarrow X$
Traités $\leftarrow \emptyset$
tant que *A_ Traiter* $\neq \emptyset$ **faire**
  choisir x dans A_Traiter tel que $d[x]$ soit minimum
  Traités $\leftarrow$ Traités $\cup \{x\}$
  A_Traiter $\leftarrow$ A_traiter - $\{x\}$
  pour tous les $y \in \Gamma^+(x)$ **faire**
    Relaxation (x, y)
retourner *le tableau d*

Procédure : Init_plus_court_chemin
Entrées : Un graphe $G = (X, A)$, un sommet s
Sorties : Un tableau d associant une valeur infinie à chaque sommet sauf s
pour tous les *sommet* x **faire**
  $d[x] \leftarrow \infty$;
$d[s] \leftarrow 0$;

Procédure : Relaxation
Entrées : un arc (x, y) de G
Sorties : Le tableau d mis à jour
si $d[y] > d[x] + w(x, y)$ **alors**
  $d[y] \leftarrow d[x] + w(x, y)$;

4.2 Plus courts chemins à partir d'une source, avec coûts arbitraires

Dans ce cas, il se peut que le graphe en antrée ait un circuit de coût strictement négatif, appelé *circuit absorbant*. Dans cette situation on ne peut pas parler de chemin de coût minimum (il y aurait des chemins infinis, de coût $-\infty$. C'est pourquoi la sortie des algorithmes sera ou bien un tableau de coûts, ou bien une sortie d'erreur en cas de détection de circuit absorbant.

Dans l'algorithme de Bellman, à l'étape K, le tableau D^K représente les longueurs des plus courts chemins de la source aux autres sommets, utilisant au plus K sommets intermédiaires.

Algorithme 5 : Bellman
Entrées : Un graphe $G = (X, A)$, un sommet s.
Sorties : Un tableau D tel que $D[x] = \sigma(s, x)$ pour tout x, ou Détéction d'un circuit absorbant.
//init
$D^0[x] \leftarrow 0$
pour tous les *sommet* $y \neq x$ **faire**
  $D^0[y] \leftarrow +\infty$
$K \leftarrow 1$
tant que $K < n$ **faire**
  $D^K[x] \leftarrow 0$
  pour tous les *sommet* y **faire**
    pour chaque $z \in \Gamma^-[x]$ **faire**
      $D^K[y] \leftarrow min(D^{K-1}[y], D^{K-1}[z] + w(x, y))$
  si $D^K = D^{K-1}$ **alors**
    retourner *Arret de la boucle*
  si $K = n + 1$ **alors**
    retourner il existe un circuit de poid négatif

4.3 Plus courts chemins entre tout couple de sommets

L'algorithme suivant dû à Floyd, très simple et de complexité $\mathcal{O}(n^3)$, calcule les plus courts chemins entre tout couple de sommets. A l'étape K, le tableau δ^k représente les congueurs des plus courts chemins de x à y dont les sommets internes sont uniquement des sommets numérotés entre 1 et K.

Algorithme 6 : Floyd

Entrées : Un graphe $G = (X, A)$, une application $w : A \leftarrow \mathbb{R}$
Sorties : Une application $\delta : X \times X \rightarrow \mathbb{R}$
//init
pour tous les $x, y \in X$ **faire**

$$\delta^0(x,y) \leftarrow \begin{cases} \infty & si \quad (x,y) \notin A \\ 0 & si \quad x = y \\ w(x,y) & si \quad (x,y) \in A \end{cases}$$

pour *K de 1 à n* **faire**
  pour tous les $x, y \in A$ **faire**
    $\delta^k(x,y) \leftarrow min(\delta^{k-1}(x,y), (\delta^{K-1}(x,x_k) + \delta^{K-1}(x_k,y)))$
si *Il existe* $x \in X$ *tel que* $\delta^n(x,x) < 0$ **alors**
  retourner circuit de cout négatif
sinon
  retourner δ^n (matrice nxn)

Chapitre 5

Arbre recouvrant de poids minimum

Kruskal et Prim

Algorithme 7 : Kruskal

Entrées : Un graphe $G = (X, A)$non orienté, une application $w : A \leftarrow \mathbb{R}$
Sorties : Un arbre[a] $T = (X, A_T)$ recouvrant de poid minimum, $A_T \subseteq A$,$w(A_T)$ et minimum)
Trier les arrêtes par poids croissants dans L
$A \leftarrow \emptyset$
tant que $L \neq 0$ **faire**
  e $\leftarrow$ tête(L)
  $L \leftarrow L \backslash \{e\}$
  si $(X, A \cup \{e\})$ *n'a pas de cycle* **alors**
    $A \leftarrow A \cup \{e\}$
retourner (X, A)

[a]Graphe connexe sans cycle

Algorithme 8 : Prim

Entrées : Un graphe $G = (X, A)$non orienté, une application $w : A \leftarrow \mathbb{R}$
Sorties : Un arbre(Graphe connexe sans cycle $T = (X, A_T)$ recouvrant de poid minimum, $A_T \subseteq A$,$w(A_T)$ et minimum)
//init
$T \leftarrow (a, \emptyset)$
$A \leftarrow \emptyset$
pour tous les $x \in X$ **faire**
  etat[x] $\leftarrow$ non-atteint
etat[x] $\leftarrow$ atteint
//boucle principale
pour i *in* $0..(n-1)$ **faire**
  choisir l'arête $e = (x, y)$ de poid minimum parmi celles ayant une extrêmité atteinte et une non atteinte
  $A \leftarrow A \cup \{e\}$
  marquer comme atteinte l'extrêmité qui était non atteinte

Chapitre 6

Flots et réseaux de transport

Un *réseau* R est un graphe orienté $G = (X, U)$, avec deux sommets particuliers s appelé *source* et p appelé *puits*, et donc chaque arc (xy) est muni d'une capacité positive $c(x, y)$. Par la suite nous supposerons que ces capacités sont des nombres naturels.

Un *flot* à travers le réseau R est une fonction associant à chaque arc (xy) une quantité $f(x, y)$, représentant la quantité d'information qui traverse l'arc. Un flot doit satisfaire deux conditions :

- Pour chaque arc (xy), la valeur du flot qui le traverse est positive et au plus égale à la capacité de l'arc : $0 \leq f(x, y) \leq c(x, y)$.
- Pour chaque sommet autre que la source et le puits, le flot rentrant dans le sommet est égal au flot sortant du sommet :

$$\forall x \in X \setminus \{s, p\}, \sum_{y \in \Gamma^-[x]} f(y, x) = \sum_{z \in \Gamma^+[z]} f(x, z)$$

Etant donné un réseau R, l'objectif est de trouver un flot qui maximise la quantité d'information transférée de la source au puits. Cette quantité, appelée *valeur* du flot, est égale au flot sortant de la source, moins le flot rentrant dans la source[1].

Vocabulaire valeur du flot f $|f|$ = flot sortant de s - flot entrant dans s

Un flot est dit bloquant si il n'existe plus de chemins non saturé

Une coupe $(Y, \bar{Y})$ de X est une partition des sommets tq $s \in Y$ et $p \in \bar{Y}$

Capacité de la coupe :

$$cap(Y, \bar{Y}) = \sum_{(X,Z) \in A,\ X \in Y,\ Z \in \bar{Y}} c(x, y)$$

Lemme 7 *Soit f un flot, soit $(Y, \bar{Y})$ une coupe,*

$$f(Y, \bar{Y}) = |f| \stackrel{def}{=} f(\{S\}, X \setminus \{S\})$$

Remarque 8 $\forall f,\ \forall (Y, \bar{Y}),\ f(Y, \bar{Y}) \leq cap(Y, \bar{Y})$

Théorème 9 *Soit f un flot tel que dans G_f il n'y a plus de chemins de S à P, Y_f : $\{x \in X | il\ existe\ un\ chemin\ de\ S\ à\ X\ dans\ G_f\}$:*

- *$|f|$ est de valeur maximum*
- *$(Y_f, \bar{Y_f})$ est de capacité minimum*
- *$f(Y_f, \bar{Y_f}) = cap(Y_f, \bar{Y_f})$*

Algorithme 9 : Ford Fulkerson

Entrées : Réseau (G, s, p, c)
Sorties : Flot de valeur maximale
// init
$f \leftarrow 0$;
$G_f \leftarrow G$;
tant que $\exists\ \mu$: *chemine de S à P dans G_f* **faire**
  f' *flot envoyant cap(μ) unit a trvers* μ ;
  $f \leftarrow f + f'$;
  Mise à Jour(G_f) ;
Retourner f ;

[1] Dans la pratique, les algorithmes ne font jamais rentrer du flot dans la source.

Deuxième partie

Algorithmes un peu plus avancés

Chapitre 7

Graphes planaires

7.1 Algorithme de reconnaissance

Etant donné un graphe $G = (X, A)$ deux-connexe, on veut vérifier si G est planaire ou non. Remarquer que la reconnaissance de la planarité pour les graphes quelconques ramène facilement à la reconnaissance de la planarité pour les graphes deux-connexes !

On considère un sous-graphe $G' = (X', A')$ de G. Au cours de l'algorithme, G' correspondera à la partie "déjà dessinée" du graphe G.

Définition 10 *Soient $G = (X, A)$ et $G' = (X', A')$ deux graphes tels que $G' \subseteq G$. On appelle* pont *du graphe G par rapport à G' :*

- *Chaque arête xy de G telle que $x, y \in V'$ et $xy \notin A'$. Dans ce cas, on appelle* pieds *du pont les sommets x et y.*
- *Chaque graphe de la forme $G(C) = (C \cup N(C), A(C))$ où C est une composante connexe de $G[V - V']$ et $N(C)$ est le voisinage de C dans G. Les arêtes de $G(C)$ sont les arêtes de G ayant au moins une extrémité dans C. Dans ce cas, on appelle* pieds *du pont les sommets de $N(C)$.*

Si G' est planaire et P est un pont de G par rapport à G', on dit qu'une face de G' est *compatible* avec le pont P si tous les pieds du pont sont sur cette face. Intuitivement, ceci veut dire que le pont P peut eventuellement être dessiné à l'intérieur de cette face.

L'algorithme 10, testant la planarité d'un graphe deux-connexe, est dû à Demoucron, Malgrange et Petruiset.

Algorithme 10 : Planarité

Entrées : un graphe G deux-connexe
Sorties : VRAI si G est planaire, $FAUX$ sinon

$G' \leftarrow$ un cycle de G
$\mathcal{F} \leftarrow \{F_1, F_2\}$ les faces de G'
tant que $G' \neq G$ **faire**
  calculer les ponts de G par rapport à G'
  pour chaque *pont* P **faire**
    calculer la liste des faces de G' compatibles avec P
  si *il existe un pont qui n'est compatible avec aucune face* **alors**
    retourner $FAUX$
  sinon
    si *il existe un pont* P *compatible avec une seule face* F **alors**
      $face\_courante \leftarrow F$
      $pont\_courant \leftarrow P$
    sinon
      /* tous les ponts sont compatibles avec au moins deux faces */
      $pont\_courant \leftarrow$ un pont quelconque
      $face\_courante \leftarrow$ une face compatible avec $pont\_courant$
  choisir deux pieds x, y du $pont\_courant$ /* justifier leur existence !*/
  $\mu \leftarrow$ un chemin de x à y dans $pont\_courant$
  /* on rajoute à G' les sommets et les arêtes de μ, dans $face\_courante$*/
  $G' = G' + \mu$
  μ découpe $face\_courante$ en deux faces F' et F''
  $\mathcal{F} \leftarrow \mathcal{F} - \{face\_courante\} \cup \{F', F''\}$
retourner $VRAI$

Chapitre 8

Cycles euleriens et hamiltoniens

8.1 Graphes eulériens

Définition 11 *On considère un graphe non orienté $G = (X, A)$. Un cycle μ de G est appelé* cycle eulérien *si μ passe exactement une fois par chaque arête de G. Un graphe qui possède un cycle eulérien est appelé* graphe eulérien

Théorème 12 *Le graphe $G = (X, A)$ est eulérien si et seulement si chaque sommet de G est de degré pair.*

La condition est nécessaire puisque si μ est un cycle eulérien de G et que μ rencontre k fois un sommet x, alors x est incident a exactement k arêtes (μ rentre k fois dans x, et il en ressort k fois aussi).

L'algorithme 11 montre que si tous les sommets sont de degré pair, le graphe est eulérien.

Algorithme 11 : CycleEulerien

Entrées : $G = (X, A)$ dont tous les sommets sont de degré pair
Sorties : un cycle eulérien μ_e de G

partir d'un sommet quelconque x de G et chercher un cheminement μ maximal (que l'on ne peut pas prolonger sans passer deux fois par une même arête)
    // montrer que μ est un cycle, en utilisant la parité des degrés !
$G' = G - artes(\mu)$
pour chaque *composante connexe C_i de G' t.q. $|C_1| \geq 2$* **faire**
    $\mu_i =$ `CycleEulerien`$(G'[C_i])$
        // remarquer que $G[C_i]$ n'a que des sommets de degré pair
    soit x_i un sommet appartenant à μ et μ_i
        // montrer que x_i existe !
μ_e est obtenu en "collant" μ et les cycles μ_i, via les sommets x_i
    // remarquer que μ et les μ_i couvrent toutes les arêtes de G
retourner μ_e

8.2 Problème du postier chinois

Le problème du postier chinois est le suivant : étant donné un graphe non orienté $G = (X, A)$ et une fonction de coût $tc : E \to \mathbb{R}^+$, trouver un cycle μ passant au moins une fois par

chaque arête du graphe et tel que le coût de μ soit le plus petit possible.

On peut imaginer que le graphe représente une ville : les arêtes sont des rues, les sommets sont des carrefours, et le postier doit passer au moins une fois par chaque rue, tout en minimisant le chemin total parcouru.

Ce problème peut être résolu en temps $\mathcal{O}(n^3)$, par l'algorithme suivant.

Algorithme 12 : PostierChinois

Entrées : le graphe $G = (X, A)$, un cout $c(xy)$ réel positif associé à chaque arête xy
Sorties : un cycle μ passant au moins une fois par chaque arête du graphe tel que μ soit de coût minimum

si *chaque sommet de G est de degré pair* **alors**
  // G est eulerien
  $\mu \leftarrow CycleEulerien(G)$
  retourner μ
// G n'est pas eulérien, il faut s'occuper des
// sommets de degré impair pour le "rendre eulérien"
$Impairs \leftarrow \{x \in X | x$ est de degré impair$\}$
soit G' une clique ayant comme ensemble de sommets l'ensemble Impairs
pour chaque $x, y \in Impairs$ **faire**
  $\mu(x, y)$ est un plus court chemin de x à y dans G
  $c'(x, y) \leftarrow longueur(\mu(x, y))$
calculer un couplage parfait M de G', tel que M soit de coût minimum pour la fonction de coût c'
pour chaque *arête xy du couplage M* **faire**
  dupliquer, dans G, le chemin $\mu(x, y)$
  // on duplique chaque arête de $\mu(xy)$,
  // donc G devient un multi-graphe
$\mu \leftarrow CycleEulerien(G)$
retourner μ

8.3 Graphes hamiltoniens

Définition 13 *On considère un graphe non orienté $G = (X, A)$. Un cycle μ de G est appelé* cycle hamiltonien *si μ passe exactement une fois par chaque sommet de G. Un graphe qui possède un cycle hamiltonien est appelé* graphe hamiltonien

Contrairement au cas des graphes eulériens, nous n'avons pas de propriété simple qui permette de vérifier si un graphe est hamiltonien ou pas. Le problème de l'hamiltonicité (étant donné G, est-il hamiltonien) est un problème NP-complet.

8.3.1 Problème du voyageur de commerce

Un voyageur de commerce doit parcourir chaque grande ville du pays et retourner au point de départ, tout en minimisant le chemin parcouru. Le pays est représenté par un graphe $G = (X, A)$. Les sommets sont les villes, les arêtes correspondent aux routes entre deux sommets, chaque arête xy possède un coût $c(x, y)$ positif indiquant la longueur de la route. Le problème du voyageur de commerce, noté TSP (comme travelling salesman's problem) consiste

à chercher un cycle μ de G, passant au moins une fois par chaque sommet, et qui soit de coût minimum.

Il est facile de prouver que le problème du voyageur de commerce est NP-difficile, en utilisant le fait que le problème de l'hamiltonicité est NP-difficile (si, si, essayez !)

L'algorithme 13 donne une 2-approximation du problème du voyageur de commerce. Cet algorithme calcule simplement un arbre recouvrant de coût minimum de G, et le cycle renvoyé est un simple parcours en profondeur de cet arbre.

Algorithme 13 : TSP_2-approximation

Entrées : le graphe $G = (X, A)$, un cout $c(xy)$ réel positif associé à chaque arête xy
Sorties : un cycle μ passant au moins une fois par chaque sommet du graphe tel que μ soit de coût au plus égal à deux fois le coût d'un cycle optimal

calculer un arbre recouvrant T de coût minimum de G
retourner le cycle correspondant au parcours en profondeur de T

Théorème 14 *L'algorithme 13 est une 2-approximation du problème du voyageur de commerce.*

Preuve. Soit T l'arbre recouvrant de coût minimum de G calculé par l'algorithme 13 et soit μ_{OPT} un cycle passant au moins une fois par chaque sommet de G, de coût minimum. Le graphe $G[\mu_{OPT}]$, induit dans G par les arêtes de μ_{OPT}, est connexe. Il possède donc un arbre recouvrant T'. Clairement T' est un arbre recouvrant de G, et $c(T') \leq c(\mu_{OPT})$. Puisque T est un arbre recouvrant de coût minimum de G, on a $c(T) \leq c(T')$, donc $c(T) \leq c(\mu_{OPT})$.

Le cycle μ renvoyé par notre algorithme consiste en un parcours en profondeur de T, donc le cout de μ est exactement deux fois le coût de T. On en conclut que $c(\mu) = 2c(T) \leq 2c(\mu_{OPT})$, donc μ est un cycle au pire deux fois plus coûteux que le cycle optimal. $\diamond$

Une technique plus élaborée nous permet d'obtenir une $\frac{3}{2}$ approximation du problème du voyageur de commerce. On commence comme précédemment par calculer un arbre recouvrant de coût minimum T de G. Au lieu de renvoyer simplement un cycle correspondant au parcours de cet arbre, l'idée de l'algorithme 14 est de prolonger T en un graphe éulérien (noté G'' das l'algorithme) obtenu en rajoutant certains chemins à T. Ces chemins sont choisis de manière très similaire à la méthode utilisée dans la résolution du problème du postier chinois.

Algorithme 14 : TSP_1.5-approximation

Entrées : le graphe $G = (X, A)$, un cout $c(xy)$ réel positif associé à chaque arête xy
Sorties : un cycle μ passant au moins une fois par chaque sommet du graphe tel que μ soit de coût au plus égal à $\frac{3}{2}$ fois le coût d'un cycle optimal

calculer un arbre recouvrant T de poids minimum de G
// on veut "rendre eulérien" l'arbre T, en lui rajoutant des arêtes
$Impairs \leftarrow \{x \in X | x$ est de degré impair dans l'arbre $T\}$
// $Impairs$ est l'ensemble des sommets de degré impair **de l'arbre T !**
soit G' une clique ayant comme ensemble de sommets l'ensemble Impairs
pour chaque $x, y \in Impairs$ **faire**
  $\mu(x, y)$ est un plus court chemin de x à y dans G
  $c'(x, y) \leftarrow longueur(\mu(x, y))$
calculer un couplage parfait M de G', tel que M soit de coût minimum pour la fonction de coût c'
// construction, à partir de l'arbre T, d'un graphe eulérien G''
$G'' \leftarrow T$
pour chaque *arête xy du couplage M* **faire**
  $G'' \leftarrow G'' \cup \mu(x, y)$
  // on rajoute à G'' une copie du chemin $\mu(x, y)$
  // G'' devient un multi-graphe
$\mu \leftarrow CycleEulerien(G'')$
retourner μ

Lemme 15 *Le coût du couplega M choisi par l'algorithme 14 est au plus $\frac{1}{2}c(\mu_{OPT})$, où μ_{OPT} est un cycle passant au moins une fois par chaque sommet de G, de coût minimum.*

Preuve. Considérons que μ_{OPT} possède une origine et un sens. Soient $x_1, \ldots, x_{2k}$ les sommets de l'ensemble $Impairs$, dans l'ordre dans lequel ils sont rencontrés par μ_{OPT} (on ne compte que la première occurence de chaque sommet sur le cycle). On note $\mu_{i,j}$ le sous-chemin de x_i à x_j dans μ_{OPT}, dans le sens de μ_{OPT}. Soient $M_1 = \{\mu_{1,2}, \mu 3, 4, \ldots, \mu_{2k-1,2k}\}$ et $M_2 = \{\mu_{2,3}, \mu 4, 5, \ldots, \mu_{2k,1}\}$. Clairement, $c(\mu_{OPT}) = c(M_1) + c(M_2)$, donc $c(M_1) \leq \frac{1}{2}c(\mu_{OPT})$ ou $c(M_2) \leq \frac{1}{2}c(\mu_{OPT})$. Remarquer que le coût du couplage M calculé par notre algorithme ne peut pas dépasser $c(M_1)$, ni $c(M_2)$. Donc $c'(M) \leq \frac{1}{2}c(\mu_{OPT})$.

Rappelons aussi que $c(T) \leq c(\mu_{OPT})$. Il s'ensuit que le coût du chemin μ renvoyé par l'algorithme est $c_(\mu) = c(T) + c'(M) \leq \frac{3}{2}c(\mu_{OPT})$. ◇

Chapitre 9

Coloration. Stable de cardinal maximum.

Définition 16 *Soit $G = (X, A)$ un graphe non orienté. On appelle* k-coloration *de G une fonction $c : V \to \{1, \ldots, k\}$ qui associe à chaque sommet une couleur parmi $\{1, \ldots, k\}$, ayant la propriété que deux sommets adjacents ne peuvent pas avoir la même couleur : $\forall xy \in E$, $c(x) \neq c(y)$.*

Un graphe G est dit k-colorable *si G possède une k-coloration. Le* nombre chromatique *de G, noté $\chi(G)$, est le plus petit entier k tel que G soit k-colorable.*

Le *problème de la coloration* est le suivant : étant donné un graphe $G = (X, A)$, trouver $\chi(G)$ et une coloration optimale de G (avec $\chi(G)$-couleurs).

Définition 17 *On rappelle qu'une* clique *de $G = (X, A)$ est un ensemble de sommets K, deux à deux adjacents : $\forall x, y \in K, xy \in E$. Le* nombre de clique *de G est la taille de la clique de cardinal maximum de G :*

$$\omega(G) = \max_{K \text{ clique de } G} |K|$$

Un stable *de G est un ensemble S de sommets deux à deux non adjacents : $\forall x, y \in S, xy \notin E$. Le* nombre de stabilité *de G est la taille du stable de cardinal maximum de G :*

$$\alpha(G) = \max_{S \text{ stable de } G} |S|$$

Le *problème du stable (resp. de la clique) de cardinal maximum* consiste à prendre en entrée un graphe $G = (X, A)$ et a calculer un stable (resp une clique) de cardinal maximum de G.

Les problèmes de la coloration, du stable de cardinal maximum et de la clique de cardinal maximum sont NP-complets. Contrairement au problème du voyageur de commerce, il a été montré que ces problèmes ne sont pas approximables à une constante près (sauf si $P = NP$, ou quelque chose de similaire). En fait, des résultats relativement récents indiquent qu'aucune approximation "raisonnable" n'est possible pour ces trois problèmes.

Autrement dit, les seuls moyens d'aborder ces questions consistent à :
- appliquer des heuristiques (dont la qualité du résultat n'est pas garantie)
- résoudre ces problèmes pour des classes de graphes ayant des propriétés ; particulières (or ces classes sont souvent très restreintes).

9.1 Relations entre nombre de clique, nombre de stabilité et nombre de clique

Proposition 18 *Pour tout graphe G,*

1. $\alpha(G) = \omega(\overline{G})$
2. $\omega(G) = \alpha(\overline{G})$
3. $\chi(G) \geq \omega(G)$

Preuve. Laissée au lecteur ! ◇

Le problème du stable de cardinal maximum est dont identique au problème de la clique de cardinal maximum, modulo le passage au graphe complémentaire.

On pourrait croire d'après la dernière relation que tout graphe G est coloriable avec $\omega(G)$ couleurs. Il suffit de considérér le graphe C_{2k+1} (le cycle avec $2k+1$ sommets, $k \geq 2$) pour se convaincre que ce cycle a un nombre chromatique égal à 3, alors que son nombre de clique est 2. Il existe des graphes avec le nombre de clique égal à 2 et un nombre chromatique arbitrairement grand. Le problème de la coloration et celui de la clique de cardinal maximum sont vraiment différents.

9.2 Heuristiques

9.2.1 Stable maximal et clique maximale

Définition 19 *Un stable S de G est dit* maximal par inclusion *s'il n'existe pas de stable S' de G tel que S soit strictement contenu dans S'.*

Une clique K est maximale par inclusion *s'il n'existe pas de clique K' avec $K \subset K'$.*

Clairement, un stable de cardinal maximum de G est aussi un stable maximal par inclusion. La réciproque est fausse (donner un exemple !). L'heuristique la plus classique pour tenter d'obtenir un stable de grande taille consiste à chercher un stable maximal par inclusion (algorithme 15). On peut rajouter des critères censés améliorer le comportement de l'heuristique, comme par exemple le fait de choisir à chaque étape un sommet non marqué ayant un nombre minimum de voisins non marqués.

Théorème 20 *L'algorithme 15 calcule un stable maximal du graphe en entrée.*

La preuve est laissée au lecteur. *Prouver aussi que le calcul d'un stable maximal, ainsi que le calcul d'une clique maximale, peut se faire en temps linéaire.*

Algorithme 15 : StableMaximal

Entrées : un graphe $G = (X, A)$, non orienté
Sorties : un stable maximal S de G

pour chaque *sommet x de G* **faire**
  $marque[x] \leftarrow FAUX$
$S \leftarrow \emptyset$
tant que *il existe un sommet x sommets non marqué* **faire**
  $S \leftarrow S \cup \{x\}$
  $marque[x] \leftarrow VRAI$
  pour chaque $y \in \Gamma(x)$ **faire**
    $marque[y] \leftarrow VRAI$
retourner S

9.2.2 Coloration gloutonne

L'algorithme 16 est un algorithme glouton de coloration des graphes. A chaque étape, il colorie un nouveau sommet, en lui attribuant la plus petite couleur possible, i.e. compatible avec les sommets déjà coloriés.

Bien que très simple, cet algorithme sert de base à la coloration optimale ou prèsque optimale pour certains classes de graphes. Par exemple, si le graphe G est planaire, il existe un ordre $x_1, \ldots, x_n$ sur les sommets de sorte à ce que chaque x_i ait au plus 5 voisins de la forme $x_j, j < i$ (pourquoi ?). En utilisant cet ordre en entrée, l'algorithme 16 obtient une 6-coloration de G. Aussi, pour les graphes d'intervalles (cf. exercice sur l'affectation des salles de cours), on procède par ordonner d'abord les sommets du graphe de manière convenable, et la coloration gloutonne appliquée à cet ordre permet d'obtenir une solution optimale.

Algorithme 16 : ColorationGloutonne

Entrées : un graphe $G = (X, A)$, non orienté et un ordre $(x_1, x_2, \ldots, x_n)$ sur les sommets de G
Sorties : une coloration de G

pour *i de 1 à n* **faire**
  donner à x_i la plus petite couleur qui n'a pas été utilisée par ses voisins
retourner *la coloration obtenue*

Théorème 21 *L'algorithme 16 donne une coloration correcte du graphe en entrée. Cet algorithme est de complexité linéaire.*

Prouver ce théorème et proposer des variantes de l'heuristique gloutonne, qui vous semblent obtenir une meilleurs coloration.