# Petit cours d'algorithmique des graphes

I. Todinca

25 mars 2005

# 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 $x_i x_{i+1} \in A$, $\forall i, 1 \leq i \leq p - 1$.*

*...*

## 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 2.1.

### 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.

**Algorithme 2.1 : Parcours en largeur**

```
Entrée : un graphe G = (X, A)
Sortie : un ordre σ sur les sommets et une arborescence
debut
    // Initialisation
    pour chaque x ∈ X
        etat[x] ← non_atteint
    index ← 1
    a_traiter ← ∅ // a_traiter est une FILE
    // Boucle principale
    pour chaque z ∈ X
        si etat[z] = non_atteint
            // On lance un parcours à partir de z
            pere[z] ← nil
            σ[z] ← index
            index + +
            a_traiter ← a_traiter ∪ {z}
            tant_que a_traiter ≠ ∅
                x ← Tete(a_traiter)
                pour chaque y ∈ Γ(x)
                    si etat[y] = non_atteint
                        etat[y] ← atteint
                        pere[y] ← x
                        σ[y] ← index
                        index + +
                        a_traiter ← a_traiter ∪ {y}
                a_traiter ← a_traiter − {x}
                etat[x] ← traite
fin
```

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 2.2 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.

**Algorithme 2.2 : Parcours en profondeur**

```
Entrée : un graphe G = (X, A)
Sortie : un ordre σ sur les sommets et une arborescence

var index : entier

procedure parcours_prof_recursif(sommet x)
debut
    etat[x] ← atteint
    σ[x] ← index
    index++
    pour chaque y ∈ Γ(x)
        si etat[y] = non_atteint
            pere[y] ← x
            parcours_prof_recursif(y)
    etat[x] ← traite
fin

debut
    // Initialisation
    pour chaque x ∈ X
        etat[x] ← non_atteint
    index ← 1
    // Boucle principale
    pour chaque x ∈ X
        si etat[x] = non_atteint
            // On lance un parcours à partir de x
            pere[x] ← nil
            parcours_prof_recursif(x)
fin
```

### 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

# Chapitre 4

# Plus courts chemins dans un graphs

## Chapitre 5

# Arbre recouvrent de poids minimum

## Chapitre 6

# Flots et réseaux de transport

# 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 6** *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 7.1, testant la planarité d'un graphe deux-connexe, est dû à Demoucron, Malgrange et Petruiset.

**Algorithme 7.1 : Planarité**

```
Entrée : un graphe G deux-connexe
Sortie : VRAI si G est planaire, FAUX sinon
begin
    G' ← un cycle de G
    F ← {F1, F2} les faces de G'
    tantque G' ≠ G faire
        calculer les ponts de G par rapport à G'
        pour chaque pont P
            calculer la liste des faces de G' compatibles avec P
        si il existe un pont qui n'est compatible avec aucune face
            retourner FAUX
        sinon si il existe un pont P compatible avec une seule face F
            face_courante ← F
            pont_courant ← P
        sinon /* tous les ponts sont compatibles avec au moins deux faces */
            pont_courant ← un pont quelconque
            face_courante ← une face compatible avec pont_courant
        choisir deux pieds x, y du pont_courant /* justifier leur existance !*/
        μ ← 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' + μ
        μ découpe face_courante en deux faces F' et F''
        F ← F − {face_courante} ∪ {F', F''}
    fin_tantque
    retourner VRAI
end
```

# Chapitre 8

# Cycles euleriens et hamiltoniens

## 8.1 Graphes eulériens

**Définition 7** *On considère un graphe non orienté $G = (X, A)$. Un cycle $\mu$ de $G$ est appelé* cycle eulérien *si $\mu$ passe exactement une fois par chaque arête de $G$. Un graphe qui possède un cycle eulérien est appelé* graphe eulérien

**Théorème 8** *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 $\mu$ est un cycle eulérien de $G$ et que $\mu$ rencontre $k$ fois un sommet $x$, alors $x$ est incident a exactement $k$ arêtes ($\mu$ rentre $k$ fois dans $x$, et il en ressort $k$ fois aussi).

L'algorithme 8.1 montre que si tous les sommets sont de degré pair, le graphe est eulérien.

## 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 $\mu$ passant au moins une fois par chaque arête du graphe et tel que le coût de $\mu$ 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.

## 8.3 Graphes hamiltoniens

**Définition 9** *On considère un graphe non orienté $G = (X, A)$. Un cycle $\mu$ de $G$ est appelé* cycle hamiltonien *si $\mu$ 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.

**Algorithme 8.1 : CycleEulerien**

`Entrée` : $G = (X, A)$ dont tous les sommets sont de degré pair
`Sortie` : `un cycle eulérien` $\mu_e$ `de` $G$

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

**Algorithme 8.2 : PostierChinois**

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

`si` chaque sommet de $G$ est de degré pair
  // $G$ est eulerien
  $\mu \leftarrow CycleEulerien(G)$
  `retourner` $\mu$
// $G$ 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 $x, y \in Impairs$
  $\mu(x, y)$ 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 $M$
  dupliquer, dans $G$, le chemin $\mu(x, y)$
  // on duplique chaque arête de $\mu(x, y)$,
  // donc $G$ devient un multi-graphe
$\mu \leftarrow CycleEulerien(G)$
`retourner` $\mu$

**Algorithme 8.3 : TSP_2-approximation**

`Entrée` : le graphe $G = (X, A)$, un cout $c(xy)$ réel positif
associé à chaque arête $xy$
`Sortie` : un cycle $\mu$ passant au moins une fois
par chaque sommet du graphe tel que $\mu$ 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 $T$

### 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 $\mu$ 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 8.3 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.

**Théorème 10** *L'algorithme 8.3 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 8.3 et soit $\mu_{OPT}$ 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 $\mu_{OPT}$, 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 $\mu$ renvoyé par notre algorithme consiste en un parcours en profondeur de $T$, donc le cout de $\mu$ est exactement deux fois le coût de $T$. On en conclut que $c(\mu) = 2c(T) \leq 2c(\mu_{OPT})$, donc $\mu$ est un cycle au pire deux fois plus coûteux que le cycle optimal. ◇

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 8.4 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.

**Lemme 11** *Le coût du couplega $M$ choisi par l'algorithme 8.4 est au plus $\frac{1}{2}c(\mu_{OPT})$, où $\mu_{OPT}$ est un cycle passant au moins une fois par chaque sommet de $G$, de coût minimum.*

**Algorithme 8.4 : TSP_1.5-approximation**

`Entrée` : le graphe $G = (X, A)$, un cout $c(xy)$ réel positif
associé à chaque arête $xy$
`Sortie` : un cycle $\mu$ passant au moins une fois
par chaque sommet du graphe tel que $\mu$ 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 $x, y \in Impairs$
    $\mu(x, y)$ 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 $M$
    $G'' \leftarrow G'' \cup \mu(x, y)$
    // on rajoute à $G''$ une copie du chemin $\mu(x, y)$
    // $G''$ devient un multi-graphe
$\mu \leftarrow CycleEulerien(G'')$
`retourner` $\mu$

*Preuve.* Considérons que $\mu_{OPT}$ 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 $\mu_{OPT}$ (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 $\mu_{OPT}$, dans le sens de $\mu_{OPT}$. 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 $\mu$ 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. Clique de cardinal maximum.

**Définition 12** *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 13** *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 14** *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 maximnal et clique maximale

**Définition 15** *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 9.1). 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 16** *L'algorithme 9.1 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.*

### 9.2.2 Coloration gloutonne

L'algorithme 9.2 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

**Algorithme 9.1 : StableMaximal**

`Entrée` : un graphe $G = (X, A)$, non orienté
`Sortie` : un stable maximal $S$ de $G$

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

**Algorithme 9.2 : ColorationGloutonne**

`Entrée` : un graphe $G = (X, A)$, non orienté
  un ordre $x_1, x_2, \ldots, x_n$ sur les sommets de $G$
`Sortie` : 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

forme $x_j, j < i$ (pourquoi ?). En utilisant cet ordre en entrée, l'algorithme 9.2 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.

**Théorème 17** *L'algorithme 9.2 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.*