# Allocation de Fréquences par Coloration de Graphes

## Table des matières

**Introduction** ..... 2

**1 Généralités** ..... 2
- 1.1 Mise en évidence du phénomène d'interférence ..... 2
- 1.2 Graphes, définitions ..... 5
- 1.3 Le problème de coloration ..... 7
  - 1.3.1 Définition ..... 7
  - 1.3.2 Complexité ..... 7
- 1.4 Quelques propriétés de $\chi(G)$ ..... 8
  - 1.4.1 Encadrements ..... 8
  - 1.4.2 Graphes de Mycielski ..... 9

**2 Graphes quelconques : des algorithmes d'approximation** ..... 11
- 2.1 Coloration séquentielle : l'algorithme glouton ..... 11
- 2.2 Une première heuristique, Welsh & Powell ..... 12
- 2.3 Deuxième heuristique, DSATUR ..... 13

**3 Graphes triangulés : un algorithme linéaire de résolution exact** ..... 17
- 3.1 Généralités sur les graphes triangulés ..... 17
  - 3.1.1 Définitions, propriétés ..... 17
  - 3.1.2 Une caractérisation des graphes triangulés ..... 18
  - 3.1.3 Coloration séquentielle ..... 20
- 3.2 Rappel : Parcours en largeur – BFS. ..... 20
- 3.3 Mots et Ordre Lexicographique ..... 23
- 3.4 Parcours en largeur lexicographique – Lex_BFS ..... 25
- 3.5 Affinage de partition, implémentation ..... 28
- 3.6 Reconnaissance de graphes triangulés ..... 30

**4 Résultats expérimentaux** ..... 32
- 4.1 Complexité temporelle des algorithmes ..... 32
- 4.2 Efficacité moyenne ..... 34
- 4.3 Le benchmark de DIMACS ..... 35

**Conclusion et perspectives** ..... 38

**Bibliographie** ..... 38

**Webographie** ..... 38

# Introduction

Avec le développement des réseaux de télécommunication modernes, on assiste au déploiement d'un nombre sans cesse croissant d'antennes-relai à travers le pays. Il est donc important de pouvoir assurer le transport de l'information entre les différentes antennes, afin de satisfaire aux différents acteurs du marché : les opérateurs doivent être en mesure de transmettre les communications de leurs clients respectifs, sans se gêner entre eux, le tout en minimisant le coût de ces opérations.

Ce problème est généralement qualifié de "Problème d'Allocation de Fréquence" : étant donné un réseau d'antenne, il faut pouvoir allouer différentes fréquences aux antennes émettrices et réceptrices de sorte qu'elles puissent véhiculer de l'information sans interférer. On s'intéresse ici au cas d'une allocation simple de fréquence, et on montrera dans quelle mesure il est possible de résoudre le problème grâce au formalisme de la théorie des graphes.

Pour cela on étudiera dans un premier temps comment modéliser le problème d'allocation de fréquences, après avoir mis en évidence le phénomène d'interférence. Dans un deuxième temps il s'agira d'exposer quelques méthodes de résolution approchées, puis d'établir un algorithme linéaire de résolution exacte dans le cas très particulier des graphes triangulés. Enfin on terminera par une étude comparative des résultats expérimentaux obtenus avec les divers algorithmes exposés.

# 1 Généralités

## 1.1 Mise en évidence du phénomène d'interférence

On montre ici par une modélisation simple comment la réception de plusieurs signaux par une même antenne peut entrainer un brouillage de l'information, empêchant la réceptrice de capter correctement le signal qui lui est destiné.

Le signal sonore étant une perturbation du milieu ambiant qui se propage (variation de la pression dans l'air par exemple), on s'intéresse généralement à l'amplitude d'une telle perturbation. En radio AM par exemple (AM pour *Amplitude Modulation*), le signal audio $u_a$ module un signal de plus haute fréquence appelé "porteuse", signal sinusoïdal de fréquence $f_p$. Le signal de plus haute fréquence transportant plus d'énergie, il va voyager plus facilement. On obtient alors un *signal modulé* de la forme $u(t) = U\big(1 + \alpha u_a(t)\big)\cos\big(2\pi f_p t\big)$, avec $0 \leq \alpha \|u_a\|_\infty \leq 1$ (cf. **Fig 1.1**).

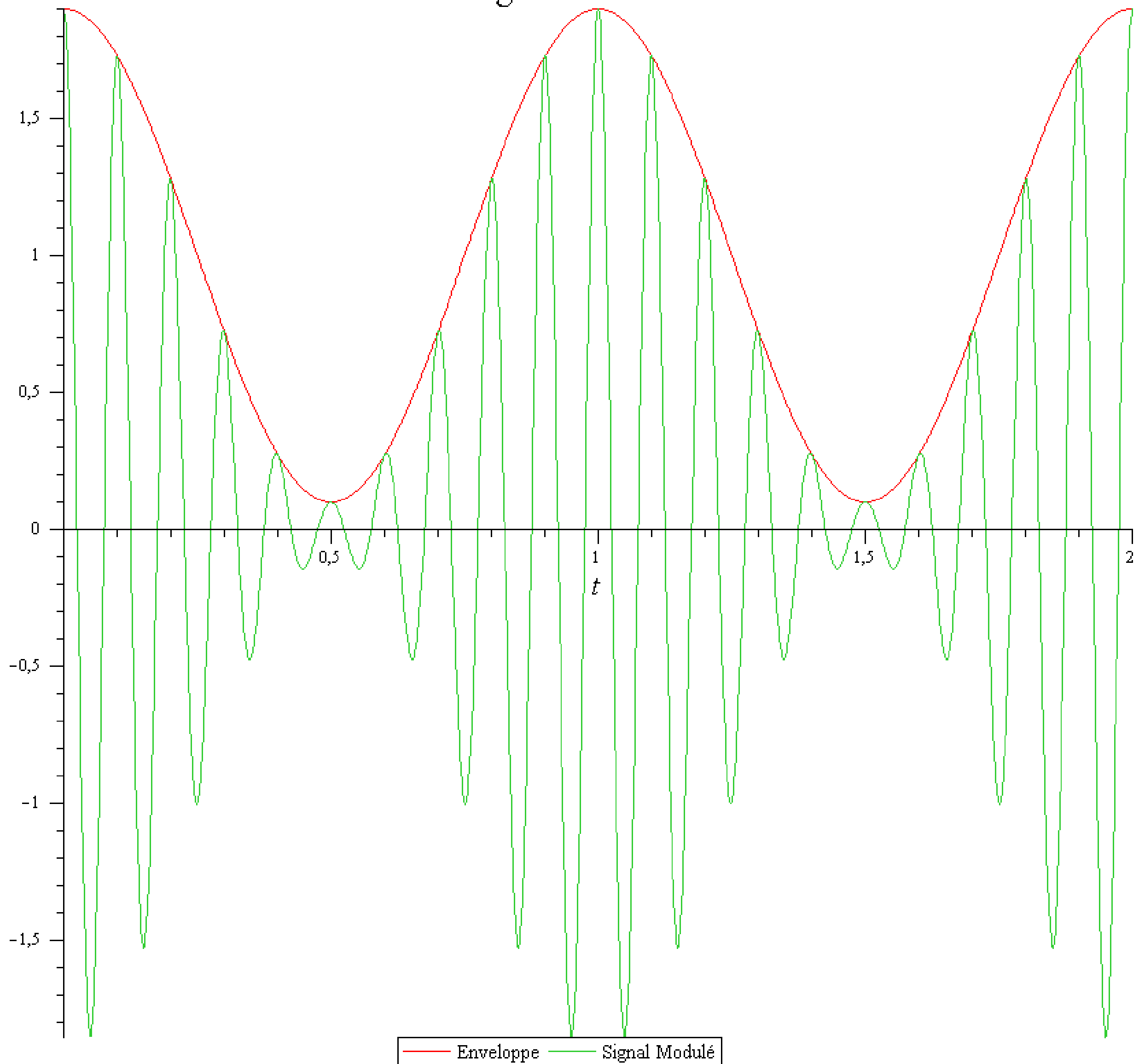



**Fig 1.1**

Pour un signal audio sinusoïdal $u_a(t) = U_a \cos(2\pi f_a t)$, où $0 \leq \alpha U_a \leq 1$, on aura donc :

$$u(t) = U(1 + \alpha U_a \cos(2\pi f_a t)) \cos(2\pi f_p t)$$
$$= U \cos(2\pi f_p t) + \frac{1}{2}\alpha U U_a \cos(2\pi(f_p + f_a)t) + \frac{1}{2}\alpha U U_a \cos(2\pi(f_p - f_a)t)$$

On a donc 3 composantes de fréquences $f_p - f_a$, $f_p$ et $f_p + f_a$, d'amplitudes respectives $\frac{1}{2}\alpha U U_a$, $U$ et $\frac{1}{2}\alpha U U_a$. D'où le spectre en fréquences représenté dans la **Fig. 1.2**. Un signal audio réel étant un signal complexe composé d'un grand nombre de composantes sinusoïdales de fréquences variant généralement entre 50 $Hz$ et 10 $kHz$, on obtiendra un spectre similaire à celui de la **Fig. 1.3**.

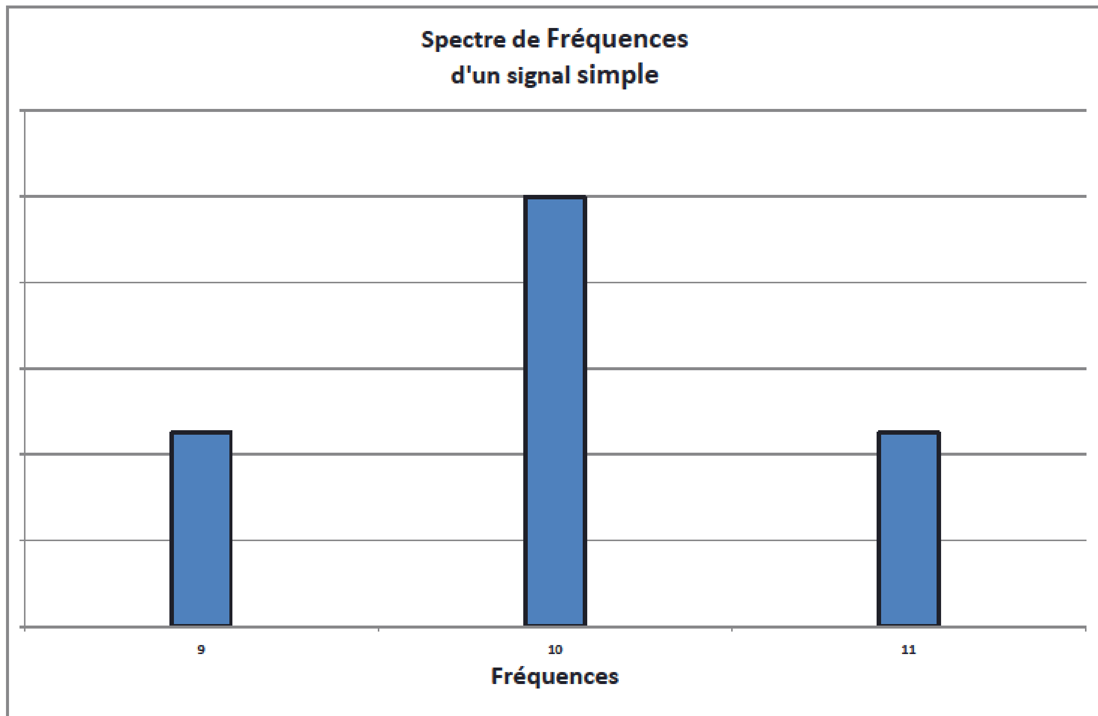



Fig 1.2

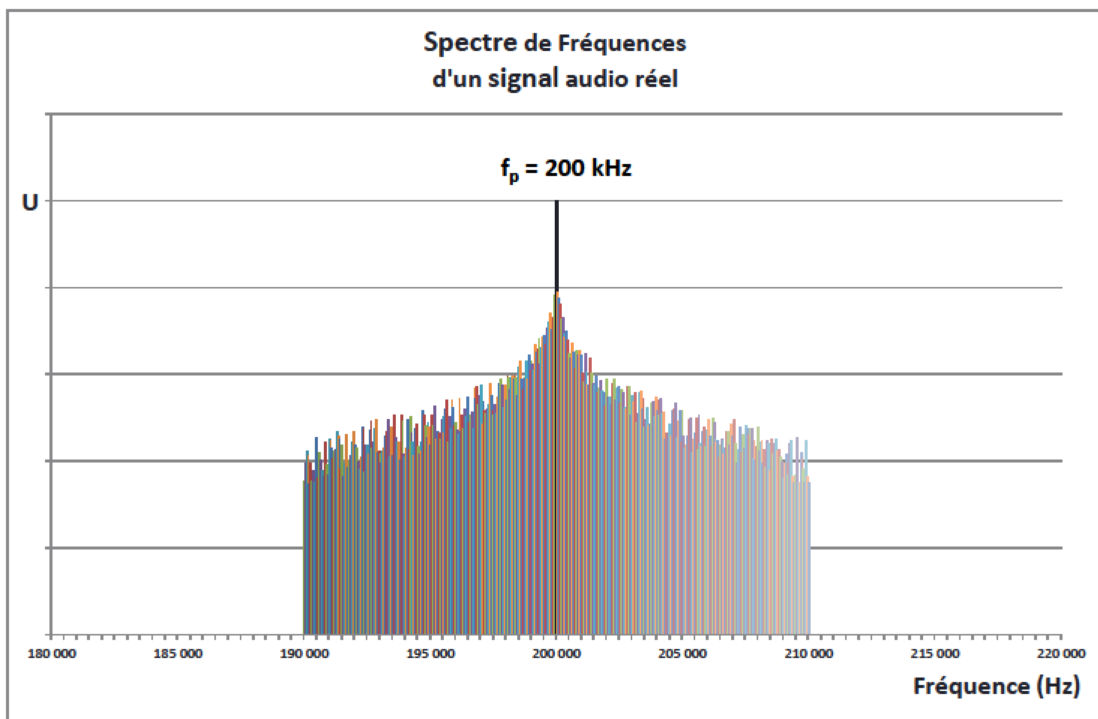



Fig 1.3

Ainsi lorsqu'un récepteur reçoit deux signaux audio dont les spectres se recouvrent, il y a brouillage de l'information et impossibilité séparer les signaux par filtrage. On voit donc que la séparation n'est possible que si l'écart entre les fréquences des deux porteuses est supérieur au double de la fréquence maximale du signal audio.

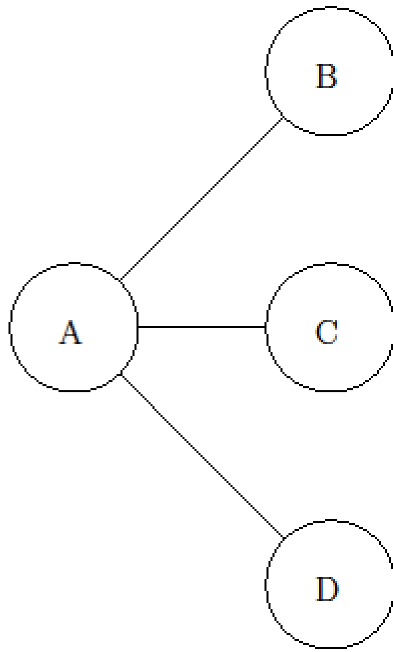
Avec notre réseau d'antenne, on considérera que deux antennes interfèrent entre elles si elles sont suffisamment proches pour que le signal émis par l'une ait une amplitude non négligeable devant celle des autres signaux reçus par l'autre (on supposera la relation "interfère" symétrique). On doit donc "réserver" des plages de fréquences différentes aux récepteurs qui interfèrent. Chaque réservation ayant un coût (financier), le problème consistera donc à minimiser le nombre de plage qu'il est nécessaire de réserver pour notre réseau afin d'assurer une communication des antennes sans interférences.

### 1.2 Graphes, définitions

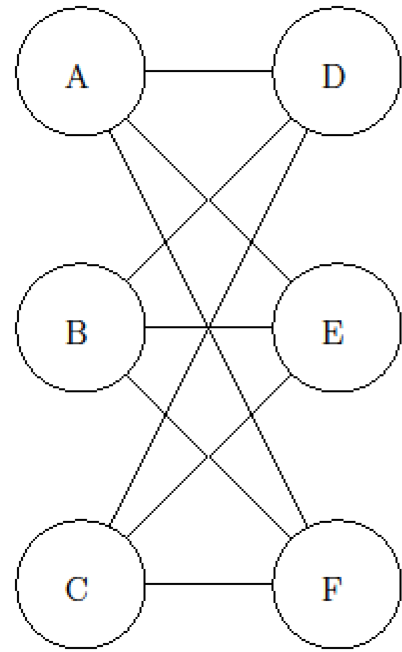
On se donne donc un réseau d'antenne, composé d'émettrices/réceptrices dont on sait si elles interfèrent ou non. On est donc amené à modéliser ce réseau par un graphe, dit *graphe d'interférence*, dont les sommets sont les antennes, que l'on relie par une arête si les antennes interfèrent. On définit ici les principales notions que nous utiliserons par la suite sur les graphes :

- Si $X$ est un ensemble fini, on notera $|X| = \text{card}(X)$ son cardinal.
- Un *graphe* est un couple $G = (V, E)$ où $V$ est un ensemble fini dont les éléments sont les *sommets* du graphe, et $E \subseteq V^2$, ses éléments sont les *arcs* du graphe. L'*ordre* (resp. la *taille*) d'un graphe est le nombre de ses sommets (resp. arêtes). On note généralement $n = |V|$ l'ordre de $G$ et $m = |E|$ sa taille. On travaillera ici sur des graphes *simples*, i.e. tels que $\forall a \in V, (a, a) \notin E$, et *non-orientés*, i.e. tels que $\forall a, b \in V, (a, b) \in E \Leftrightarrow (b, a) \in E$. Les arcs sont alors appelés des *arêtes*.
- Deux sommets $a$ et $b$ sont dit *adjacents* si $(a, b) \in E$ ou $(b, a) \in E$. Soit $a \in V$, on note alors $\mathcal{N}(a) = \{b \in V, (a, b) \in E \text{ } ou \text{ } (b, a) \in E\}$ l'ensemble des sommets adjacents à $a$, appelé le *voisinage* de $a$. On note également $d(a) = |\mathcal{N}(a)|$ le nombre de sommets adjacents à $a$, ou *degré* de $a$. Le *degré* du graphe, noté $\Delta(G)$, est le degré maximum de ses sommets.
- $H = (V', E')$ est un *sous-graphe* de $G$ si $V' \subseteq V$ et $E' \subseteq E$. Soit un ensemble $S \subseteq V$, on note $G[S]$ le graphe $(S, E \cap (S \times S))$, appelé *sous-graphe* de $G$ *induit*, ou *engendré* par $S$. Un sous-graphe $H$ de $G$ est dit *induit* si il existe $S \subseteq V$ tel que $H = G[S]$. Sinon, on dit que $H$ est *partiel*. On notera $H \subseteq G$ pour indiquer que $H$ est un sous-graphe de $G$.
- Le graphe $G$ est dit *complet* si $E = V^2 - \{(a, a), a \in V\}$ (tous les sommets sont 2 à 2 adjacents). On note $K_n$ le graphe complet d'ordre $n$ (à une renumérotation des sommets près). Notons que la taille d'un graphe complet d'ordre $n$ est $m = n(n-1)$.
- Si $G$ est un graphe simple d'ordre $n$ et de taille $m$, on définit la densité du graphe comme étant le rapport $g(G) = \frac{m}{n(n-1)} \in [0,1]$.
- On appelle *clique* un ensemble $S \subseteq V$ qui engendre un graphe complet. Le cardinal maximum d'une clique de $G$ est noté $\omega(G)$, on l'appelle parfois le *nombre de clique* de $G$.
- On appelle *stable* un ensemble $S \subseteq V$ qui engendre un graphe sans arête. Le cardinal maximum d'un stable de $G$ est noté $\alpha(G)$, appelé *nombre de stabilité* de $G$. Notons qu'un stable est aussi le complémentaire (dans $V$) d'une clique.
- Un graphe $G = (V, E)$ est dit *multiparti*, ou $p$-parti si $V$ peut être partitionné en $p$ stables. On dit que $G = (V, E)$ est multiparti complet s'il existe une partition de $V$ en $p$ stables $S_1, \dots, S_p$ telle que $\forall i, j \in [\![1, p]\!], (i \neq j) \Rightarrow (S_i \times S_j) \subseteq E$, i.e. si les sommets de différentes parties sont tous reliés entre eux. On note $K_{n_1, n_2, \dots, n_p}$ un tel graphe $p$-parti complet, où $\forall i \in [\![1, p]\!], n_i = |S_i|$.
- Une *chaîne* de *longueur* $k$ est une suite finie de sommets $v_0 v_1 \dots v_k$ telle que l'on ait $\forall i \in [\![1, k]\!], (v_{i-1}, v_i) \in E$. Un *chemin* est une chaîne $v_0 v_1 \dots v_k$ où chaque $v_i$ apparait au plus une fois. Un *cycle* de longueur $k$ est une chaîne $v_0 v_1 \dots v_k$ telle que $v_0 = v_k$. Dans une chaîne $v_0 v_1 \dots v_k$, une *corde* est une arête $(v_i, v_j) \in E$, avec $j \neq i \pm 1$. Enfin, un *trou* est un cycle sans corde. On note $C_k$ "*le*" graphe induit par un trou de longueur $k$ (à une renumérotation des sommets près).

- Soient $a, b \in V$, on dit que $b$ est *accessible* à partir de $a$ (ou depuis $a$) s'il existe un chemin qui relie $a$ à $b$, c'est-à-dire un chemin de la forme $v_0 v_1 \dots v_k$ avec $v_0 = a$ et $v_k = b$. La relation "$b$ accessible à partir de $a$", qui pourra être notée $a \dashrightarrow b$, est une relation d'équivalence (réflexive car le graphe est non orienté). On appelle *composante connexe* de $G$ une classe d'équivalence pour la relation $\dashrightarrow$. On dira que deux sommets d'une même classe sont *connectés*. Le graphe $G$ est dit *connexe* s'il possède une seule composante connexe (tous les sommets sont accessibles entre eux).

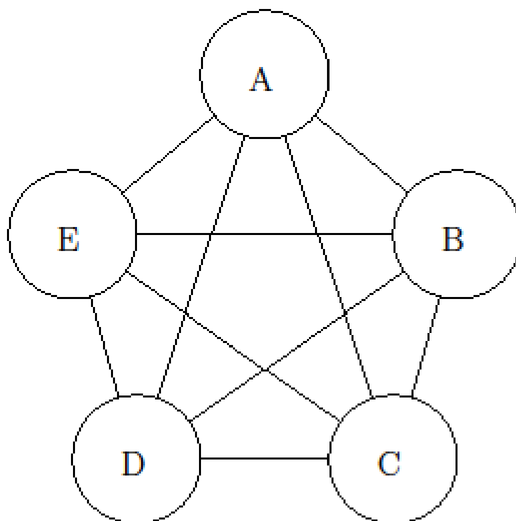



La griffe $K_{1,3}$

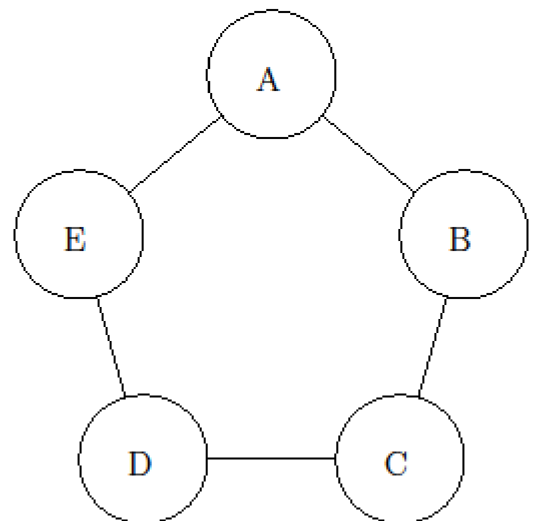



Le graphe $K_{3,3}$

**Fig 1.4**




Le graphe $K_5$




Le trou $C_5$

**Fig 1.5**

## 1.3 Le problème de coloration

### 1.3.1 Définition

On suppose maintenant connue une modélisation de notre réseau d'antennes en tant que graphe simple non-orienté. On formule ici un problème équivalent au problème d'allocation de fréquences sur un graphe simple non-orienté : celui de la coloration de graphe. Notons qu'en raison de contraintes géographiques, technologiques, ou autres (présence de montagnes, etc.), on ne peut pas faire d'hypothèses *a priori* sur la structure du graphe obtenu. Le but de ce qui suit sera donc de donner un exemple de méthode employée pour colorer un graphe, que ce soit dans le cadre de l'allocation de fréquence ou d'un problème d'une autre nature.

Soit $G = (V, E)$ un graphe simple non-orienté. Une application $\sigma : V \to \mathbb{N}$ est appelée une *coloration* de $G$. Si $a \in V$, $\sigma(a)$ est appelé *couleur* du sommet $a$. On dit que $\sigma$ est une *coloration propre* de $G$ si $\forall (a,b) \in E, \sigma(a) \neq \sigma(b)$, i.e. deux sommets adjacents n'ont jamais la même couleur. Si on note $k = |\sigma\langle V \rangle|$, on dit alors que $\sigma$ est une *$k$-coloration propre* de $G$, et que $G$ est *$k$-colorable*. Le plus petit entier $k$ pour lequel il existe une $k$-coloration propre de $G$ est appelé *nombre chromatique* de $G$, et noté $\chi(G)$. Un tel nombre existe car $G$ admet toujours une $|V|$-coloration (le graphe étant simple). Une $\chi(G)$-coloration est dite *optimale*.

Allouer des fréquences à un réseau d'antenne revient donc à chercher une coloration propre du graphe d'interférence correspondant. On numérote chaque fréquence à allouer, puis on cherchera le nombre chromatique du graphe d'interférence, ce qui minimisera le nombre de fréquences à allouer, et donc le coût de l'opération.

### 1.3.2 Complexité

On peut alors se poser la question de la complexité d'un tel problème. Sans rentrer dans les détails ou chercher à démontrer quoi que ce soit, expliquons sommairement ce qu'il en est :

- En théorie de la complexité, on dit qu'un problème appartient à la classe NP (pour *Non-déterministe Polynomiaux*) s'il peut être résolu par une machine non-déterministe en un temps polynomial. De façon équivalente, ce sont des problèmes dont on sait vérifier une solution avec une machine déterministe en un temps polynomial (on peut donc résoudre un tel problème en énumérant toutes les solutions, puis en les testant les unes après les autres …).
- Un problème $\Gamma$ est dit *NP-Difficile* si pour tout problème $\Pi$ dans NP, il existe une machine déterministe qui permette de ramener la résolution de $\Pi$ à la résolution de $\Gamma$ en un temps polynomial (on imagine bien dans ce cas que la résolution de $\Gamma$ est donc "au moins aussi difficile" que la résolution de $\Pi$, sinon $\Pi$ ne serait pas un problème de la classe NP).
- Un problème est dit *NP-Complet* s'il est NP-Difficile et appartient à la classe NP.

Il a alors été montré que déterminer le nombre chromatique d'un graphe quelconque est un problème NP-Complet. Mieux : étant donné un entier naturel $k \geq 3$, déterminer si un graphe quelconque est $k$-colorable est aussi un problème NP-Complet. Comme on n'est pas en mesure *a priori* de résoudre (par un algorithme déterministe) le problème général en un temps "raisonnable" (polynomial par exemple), on propose des algorithmes dits d'approximation, qui offrent des résultats généralement satisfaisants avec une complexité moindre. On verra également que le problème devient cependant soluble en un temps linéaire pour une certaine classe de graphe, les graphes triangulés.

## 1.4 Quelques propriétés de $\chi(G)$

### 1.4.1 Encadrements

Avec les notations introduites, on peut déjà énoncer inégalité concernant $\chi(G)$ :

**Proposition 1.4.1.a** : Soit $G$ un graphe simple non-orienté. Alors $\boxed{\chi(G) \geq \omega(G)}$

**Démonstration** : Soit $\sigma$ une coloration propre optimale et $S$ une clique de taille $\omega(G)$.
Il est alors clair que $\sigma|_S$ est injective, car, si $a, b \in S$, avec $a \neq b$, alors $(a, b) \in E$ (car $S$ est une clique), d'où $\sigma(a) \neq \sigma(b)$. On voit donc que $|\sigma\langle S\rangle| = |S| = \omega(G)$, ce qui montre bien l'inégalité car $|\sigma\langle S\rangle| \leq |\sigma\langle V\rangle| = \chi(G)$ ■

Notons que le cas d'égalité est atteint en particulier pour un graphe complet $K_n$ (on a alors $\chi(K_n) = \omega(K_n) = n$), car dans le cas d'un graphe complet, une coloration propre est injective.

Toutefois cette inégalité ne renseigne généralement pas de manière précise sur $\chi(G)$, en effet on peut trouver des graphes vérifiant $\omega(G) = 2$ et de nombre chromatique arbitrairement grand, comme les graphes de Mycielski (cf. 1.4.2).

**Proposition 1.4.1.b** : Soit $G = (V, E)$ un graphe simple non-orienté. Alors $\boxed{\chi(G) \geq \frac{|V|}{\alpha(G)}}$

**Démonstration** : Notons $G = (V, E)$ et $\sigma : V \to [\![1, \chi(G)]\!]$ une coloration propre optimale.
Soit $i \in [\![1, \chi(G)]\!]$, on remarque que $\sigma^{-1}\langle\{i\}\rangle$ est un stable de $G$, donc $|\sigma^{-1}\langle\{i\}\rangle| \leq \alpha(G)$. D'où :

$$|V| = |\sigma^{-1}\langle[\![1, \chi(G)]\!]\rangle| = \sum_{i=1}^{\chi(G)} |\sigma^{-1}\langle\{i\}\rangle| \leq \sum_{i=1}^{\chi(G)} \alpha(G) = \chi(G) \cdot \alpha(G)$$

Là encore il ne s'agit pas d'un bon indicateur sur $\chi(G)$. En effet si on considère le graphe $G = (V, E)$ composé du graphe complet $K_k$ et de $n - 1$ autres sommets non adjacents, on a bien $\chi(G) = k$, mais $\frac{|V|}{\alpha(G)} = \frac{k+n-1}{n}$, qui tend vers 1 lorsque $n$ tend vers $+\infty$ ■

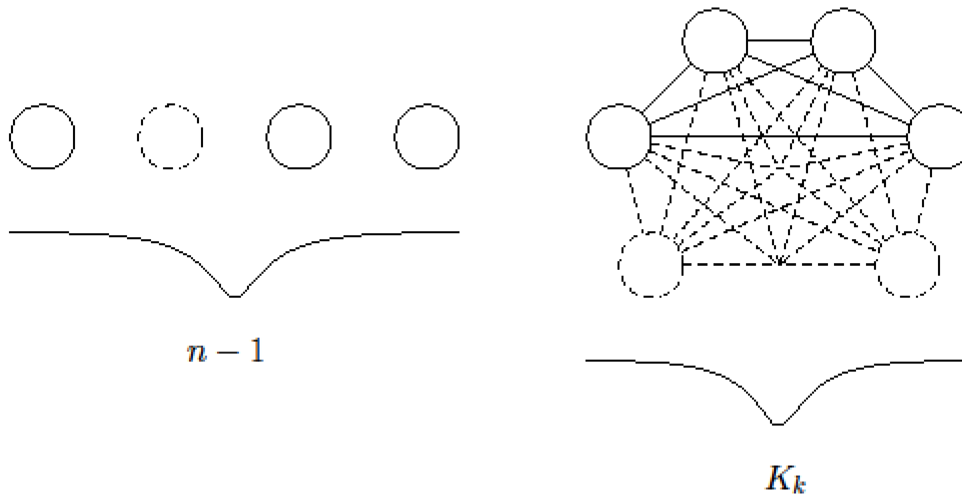



Fig 1.6

**Proposition 1.4.1.c** : Soit $G$ un graphe simple non-orienté. Alors $\boxed{\chi(G) \leq \Delta(G) + 1}$

On donnera une preuve constructive de cette inégalité avec l'algorithme de coloration séquentielle dans la partie 2.

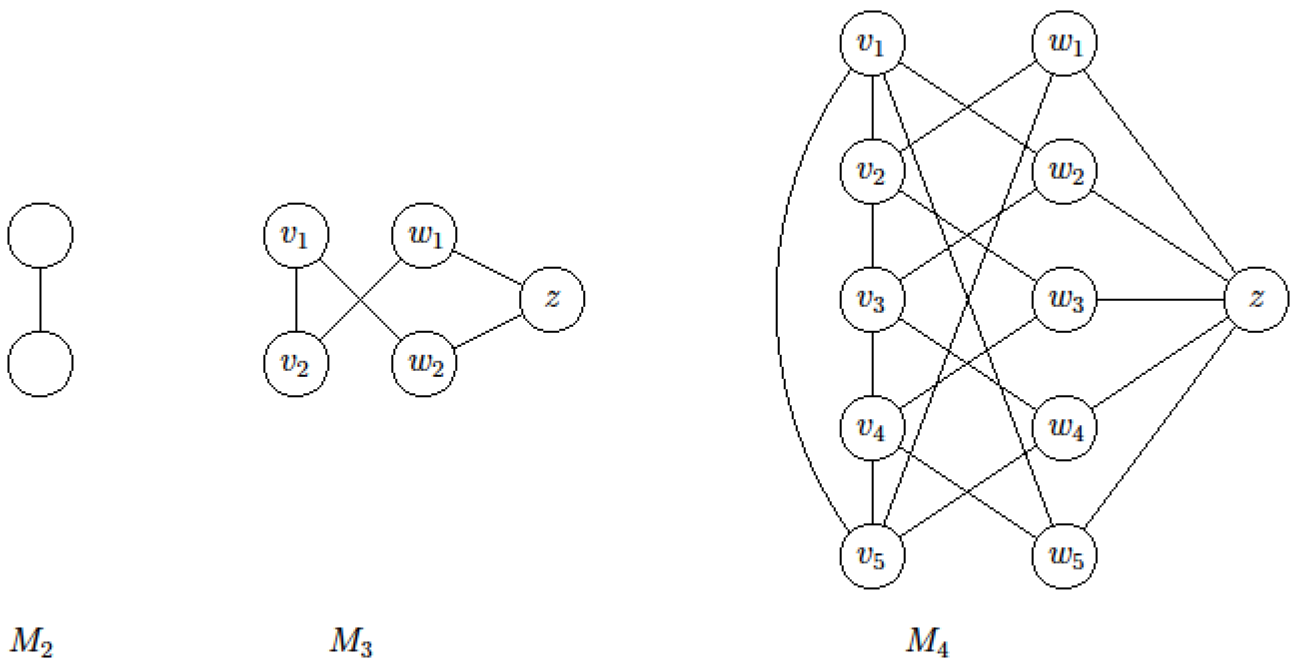
### 1.4.2 Graphes de Mycielski

**Définition** : *Graphes de Mycielski* $M_k$ $(k \geq 2)$.
Ils sont définis de manière récursive comme suit :

- $M_2 = (\{0,1\}, \{(0,1), (1,0)\})$
- Si $M_k = (V_k, E_k)$, avec $V_k = \{v_1 = 0, \dots, v_n = n-1\}$, on définit $M_{k+1} = (V_{k+1}, E_{k+1})$ par :

— $V_{k+1} = V_k \cup \{w_1 = n, \dots, w_n = 2n-1, z = 2n\}$
— $E_{k+1} = E_k \cup E' \cup E''$, avec :
— $E' = \{(w_i, v_j), (v_i, v_j) \in E_k \; et \; i, j \in [\![1, n]\!]\} \cup \{(v_i, w_j), (v_i, v_j) \in E_k \; et \; i, j \in [\![1, n]\!]\}$
— $E'' = \{(w_i, z), i \in [\![1, n]\!]\} \cup \{(z, w_j), j \in [\![1, n]\!]\}$

**Remarque** : Notons que les graphes de Mycielski sont simples, donc cette construction implique que si $(w_i, v_j) \in E_{k+1}$, alors $i \neq j$.




Graphes de Mycielski

**Fig 1.7**

**Proposition 1.4.2.a** : $\forall k \geq 2, \omega(M_k) = 2$
**Démonstration** : On procède par récurrence sur $k$.

La propriété est vraie pour $k = 2$.

Supposons maintenant que $\omega(M_k) = 2$, pour un certain $k \in \mathbb{N}$, et montrons que $\omega(M_{k+1}) = 2$. Notons $M_k = (V_k, E_k)$, où $V_k = \{v_1, \dots, v_n\}$ et $M_{k+1} = (V_{k+1}, E_{k+1})$, où $V_{k+1} = V_k \cup \{w_1, \dots, w_n, z\}$ comme défini ci-dessus. Supposons par l'absurde que $V_{k+1}$ contienne une clique de cardinal supérieur ou égal à 3. Alors $M_{k+1}$ contient un triangle $C_3$. $C_3$ ne peut être formé de trois sommets $v_{i_1}$, $v_{i_2}$ et $v_{i_3}$, car sinon on aurait $\omega(M_k) \geq 3$. Or les sommets $\{w_1, \dots, w_n\}$ sont stables, donc $C_3$ contient au plus un sommet $w_{i_0}$. De plus $\mathcal{N}(z) = \{w_1, \dots, w_n\}$, qui est stable, donc $C_3$ ne contient pas le sommet $z$. Ainsi les sommets de $C_3$ sont $w_{i_0}$ et deux autres sommets $v_{i_1}$ et $v_{i_2}$.

D'après la remarque précédente, $(w_{i_0}, v_{i_1}) \in E_{k+1}$, donc $i_0 \neq i_1$(et de même $i_0 \neq i_2$). Mais alors, par construction de $E_{k+1}$, cela signifie que $(v_{i_0}, v_{i_1}) \in E_k$ (et de même $(v_{i_0}, v_{i_2}) \in E_k$). Par symétrie on voit

finalement que $\{v_{i_0}, v_{i_1}, v_{i_2}\}$ est un clique de cardinal 3 de $M_{k+1}$, et aussi de $M_k$, ce qui contredit l'hypothèse de récurrence ■

**Proposition 1.4.2.b** : $\forall k \geq 2, \chi(M_k) = k$

**Démonstration** : On procède par récurrence sur $k$.

La propriété est vraie pour $k = 2$.

Supposons maintenant $\chi(M_k) = k$, pour un certain $k \in \mathbb{N}$, et montrons que $\chi(M_{k+1}) = k + 1$.

Notons $M_k = (V_k, E_k)$, où $V_k = \{v_1, \dots, v_n\}$ et $M_{k+1} = (V_{k+1}, E_{k+1})$, où $V_{k+1} = V_k \cup \{w_1, \dots, w_n, z\}$ comme précédemment. On dispose donc d'une coloration $\sigma : V_k \to [\![1, k]\!]$ optimale pour $M_k$. On prolonge alors $\sigma$ en une $(k+1)$-coloration $\tilde{\sigma}$ de $M_{k+1}$ en posant $\forall i \in [\![1, n]\!], \tilde{\sigma}(w_i) = \sigma(v_i)$ et $\tilde{\sigma}(z) = k + 1$.On voit alors facilement que $\tilde{\sigma}$ est une $(k+1)$-coloration propre de $M_{k+1}$, car $\forall i, j \in [\![1, n]\!], \left((w_i, v_j) \in E_{k+1}\right) \Rightarrow \left((v_i, v_j) \in E_k\right) \Rightarrow \left(\tilde{\sigma}(w_i) = \sigma(v_i) \neq \sigma(v_j) = \tilde{\sigma}(v_j)\right)$.

On vient de montrer que $\chi(M_{k+1}) \leq k + 1$. Or par construction $M_k \subseteq M_{k+1}$, donc il est clair que $k = \chi(M_k) \leq \chi(M_{k+1})$. Il reste donc à montrer que $\chi(M_{k+1}) \neq k$.

On suppose par l'absurde qu'il existe une $k$-coloration propre $\sigma$ de $M_{k+1}$. Quitte à renuméroter, on peut prendre $\sigma : V_{k+1} \to [\![1, k]\!]$ telle que $\sigma(z) = k$ (on aura alors $\forall i \in [\![1, n]\!], \sigma(w_i) \neq k$, car $\mathcal{N}(z) = \{w_i, i \in [\![1, n]\!]\}$). On définit alors une coloration de $M_k$, $\varsigma$ : $\begin{array}{ccc} V_k & \to & [\![1, k-1]\!] \\ v_i & \to & \begin{cases} \sigma(v_i) \ si \ \sigma(v_i) \neq k \\ \sigma(w_i) \ sinon \end{cases} \end{array}$

Vérifions que $\varsigma$ est une coloration propre. Soient $v_i$ et $v_j$ deux sommets adjacents de $M_k$.

- Si $\sigma(v_i) \neq k$ et $\sigma(v_j) \neq k$, alors $\varsigma(v_i) = \sigma(v_i) \neq \sigma(v_j) = \varsigma(v_j)$
- Si $\sigma(v_i) = k$, alors $\sigma(v_j) \neq k$, et $\varsigma(v_i) = \sigma(w_i) \neq \sigma(v_j) = \varsigma(v_j)$, car $(w_i, v_j) \in E_{k+1}$
- Si $\sigma(v_j) = k$, alors de même $\varsigma(v_i) \neq \varsigma(v_j)$

Finalement, on a construit une $(k-1)$-coloration de $M_k$ : contradiction avec l'hypothèse de récurrence, ce qui montre bien que $\chi(M_{k+1}) = k + 1$ ■

# 2 Graphes quelconques : des algorithmes d'approximation

## 2.1 Coloration séquentielle : l'algorithme glouton

Une première approche pour colorer le graphe est de prendre ses sommets les uns après les autres afin de leur affecter une couleur, tout en veillant à ce que deux sommets adjacents n'aient jamais la même couleur : c'est l'algorithme de coloration séquentielle. On parle d'algorithme glouton dans le sens où il consiste à faire le choix d'une couleur "optimum" localement, dans l'espoir d'obtenir une coloration finale "satisfaisante".

On peut par exemple décider de prendre pour chaque sommet la plus petite couleur non utilisée dans les voisins du sommet à colorier. L'algorithme fourni ainsi une borne supérieure pour $\chi(G)$, c'est-à-dire une coloration propre, mais pas forcément optimale. La coloration obtenue dépendra fortement de l'ordre dans lequel les sommets seront colorés. Notons ainsi que pour un graphe donné il existe toujours un ordre des sommets pour lequel l'algorithme fournisse une coloration optimale : il suffit de prendre une coloration optimale, et de numéroter les sommets par ordre croissant de couleur. Malheureusement c'est un "cercle vicieux", et l'on ne peut pas connaitre à l'avance une telle numérotation.

**NB** : Dans ce qui suit, on considère un graphe $G = (V, E)$ à colorer, et on notera systématiquement $n$ (resp. $m$) l'ordre (resp. la taille) de $G$. Les sommets du graphe seront alors des entiers numérotés de 0 à $n - 1 : V = [\![0, n - 1]\!]$. Les numérotations ou les ordres considérés sont alors des permutations de $[\![0, n - 1]\!]$ dans lui-même. De plus, si on ne le précise pas, lorsqu'il sera question de la complexité d'un algorithme, il s'agira généralement d'une complexité temporelle.

**Programme 1** Coloration séquentielle

**Entrée** – Un graphe quelconque $G = (V, E)$, une permutation des sommets $\sigma : [\![0; n - 1]\!] \to V$
**Sortie** – Une coloration propre $c : V \to \mathbb{N}$

```
1    n := |V|;
2    Couleur := Tableau (Taille : n) (Defaut : -1);
3    Pour i de 0 à n-1 faire
4          x := σ(i);
5          # Recherche de la plus petite couleur non utilisée dans N(x) #
6          Libre := Tableau (Taille : ???) (Defaut : true);
7          Pour y dans N(x) faire
8                Si Couleur.(y) ≠ -1 faire
9                      Libre.(Couleur.(y)) ← false;
10               Fin Si
11         Fin Pour;
12         index := 0;
13         Tant Que Libre.(index) = false faire
14               index := index + 1
15         Fin Tant Que;
16         # Affecter la couleur donnée par index à x #
17         Couleur.(x) ← index
18   Fin Pour;
19   Renvoyer Couleur
```

La terminaison de l'algorithme est ici triviale. Quant à sa correction, on montre facilement par récurrence sur $i < n$ que Couleur fourni bien une coloration propre du graphe $G[\sigma\langle[\![0, i]\!]\rangle]$. Le pseudo-code ci-dessus suggère également une complexité temporelle linéaire :

**Proposition 2.1.a** : La complexité temporelle d'un algorithme de coloration séquentielle est :

$$O(n+m).$$

**Démonstration** : On effectue une première boucle de $n$ itérations. Pour chaque itération, la recherche de index (lignes 6 à 15) a une complexité temporelle $O\big(d(x)\big)$, où $d(x) = |\mathcal{N}(x)|$ (nombre d'itérations de la boucle lignes 7 à 11). La complexité totale de ces opérations de recherche est donc dominée par $\sum_{x \in V} d(x) = m$. Finalement, on a bien une complexité temporelle totale $O(n+m)$ ■

**Proposition 2.1.b** : La coloration $c$ fournie par l'algorithme de coloration séquentielle vérifie l'inclusion : $c\langle V\rangle \subseteq [\![0, \Delta(G)]\!]$

**Démonstration** : On montre par récurrence sur $i$ que $c\big(\sigma(i)\big) \subseteq [\![0, \Delta(G)]\!]$.

- Pour $i = 0$, on a directement $c\big(\sigma(0)\big) = 0$.
- Supposons $c(\sigma\langle[\![0, i-1]\!]\rangle) \subseteq [\![0, \Delta(G)]\!]$ pour un certain $i \in [\![1, n-1]\!]$. On voit alors facilement

que $c\big(\sigma(i)\big) \subseteq [\![0, \Delta(G)]\!]$. En effet si on note $x = \sigma(i)$, alors $\varphi = c|_{\sigma\langle[\![0,i-1]\!]\rangle \cap \mathcal{N}(x)}$ est une fonction de $\mathcal{N}(x)$ dans $[\![0, \Delta(G)]\!]$. Mais $|\mathcal{N}(x)| = d(x) \leq \Delta(G)$, donc $\varphi$ ne peut pas être surjective, et on a donc $c(x) = \min\{k \geq 0, k \notin \varphi\langle\mathcal{N}(x)\rangle\} = \min\{\mathbb{N} \setminus \varphi\langle\mathcal{N}(x)\rangle\} \in [\![0, \Delta(G)]\!]$ ■

**NB** : Un corollaire de cette propriété est alors l'inégalité de la proposition 3 énoncée plus haut.

### 2.2 Une première heuristique, Welsh & Powell

Maintenant que l'on dispose d'un algorithme de coloration approché, il est intéressant de chercher une stratégie donnant lieu à de "bonnes" coloration. Intuitivement, on devine que les sommets possédant beaucoup de voisins seront plus "difficiles" à colorer que ceux qui en ont peu. C'est sur cette idée qu'est basé l'algorithme dit de Welsh & Powell : colorer séquentiellement les sommets du graphe par ordre décroissant de degré. On parle d'heuristique dans le sens où l'algorithme fourni une réponse approchée au problème de coloration.

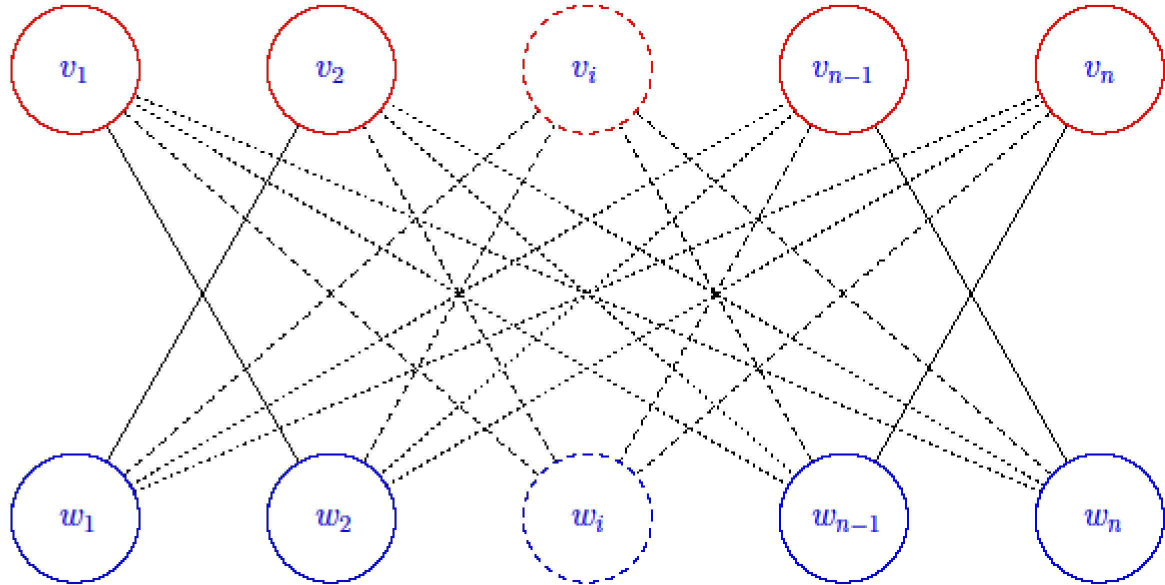
**Programme 2** Algorithme de Welsh & Powell

**Entrée** – Un graphe quelconque $G = (V, E)$
**Sortie** – Une coloration propre $c : V \to \mathbb{N}$

```
1    n := |V|;
2    Deg := Tableau (Taille : n) (Defaut : 0);
3    Sommet := Tableau (Taille : n) (Defaut : 0);
4    Pour i de 0 à n-1 faire
5        Deg.(i) ← d(i);
6        Sommet.(i) ← i
7    Fin Pour;
8    Tri (Tableau : Sommet) (Relation d'ordre : a ≼ b ssi Deg.(a) ≥ Deg.(b));
9    Coloration séquentielle (Graphe : G) (Numérotation : Sommet)
```

Si on utilise par exemple un tri par dénombrement (en $O(n)$, car les degrés de deux sommets ne peuvent différer de plus de $n$), on obtient un algorithme de complexité $O(n+m)$. Si on utilise plutôt un tri par comparaison, l'algorithme est alors de complexité $O(n\ln(n) + m)$. Cette différence est toutefois à nuancer. En effet on verra dans la dernière partie qu'en pratique le tri par fusion de la librairie Caml se révèle plus rapide qu'un tri par dénombrement, sans doute est-ce dû à une complexité spatiale plus importante.

Enfin, notons que l'on peut parfois aboutir aux pires colorations possibles avec l'algorithme de Welsh & Powell, comme on le voit sur la **Fig 2.1**. Il est néanmoins possible d'améliorer cette heuristique, c'est l'objet de l'algorithme DSATUR.




Graphes de Johnson

Fig 2.1

**Définition** : *Graphes de Johnson* $J_n$ $(n \geq 1)$.
Il s'agit du graphe biparti $G = (V, E)$ avec $V = \{v_1, \ldots, v_n, w_1, \ldots, w_n\}$ qui vérifie :
— $\forall i, j \in [\![1, n]\!], (v_i, v_j) \notin E$
— $\forall i, j \in [\![1, n]\!], (v_i, w_j) \in E \Leftrightarrow i \neq j$
C'est en fait le graphe biparti $K_{n,n}$ auquel on a enlevé les arêtes des sommets se faisant face.

Le nombre de coloration de ces graphes est évidement 2. Pourtant, si les sommets sont numérotés par exemple par $v_i = 2i$ et $w_i = 2i + 1$, on voit bien que Welsh & Powell fournira une coloration propre $c$ donnée par $c(v_i) = i - 1$ et $c(w_i) = i - 1$ (se démontre aisément par récurrence sur $n$ : tous les sommets étant de même degré, ils vont être visités dans l'ordre croissant des numéros).

### 2.3 Deuxième heuristique, DSATUR

Cette fois, on considère toujours qu'un sommet est plus "difficile" à colorer s'il a beaucoup de voisins, mais on prend également en compte le fait qu'un sommet qui possède déjà beaucoup de voisins de couleurs différentes sera aussi "problématique". On modifie alors l'algorithme de coloration séquentielle pour colorer en priorité de tels sommets. Concrètement, on définit la notion de *degré de saturation* d'un sommet $a$, notée $DSAT(a)$, calculé à un certain moment de l'algorithme comme suit :

- Si $a$ n'a aucun voisin colorié alors $DSAT(a) =$ degré de $a$.
- Sinon, $DSAT(a) =$ nombre de couleurs différentes utilisées pour colorer les voisins de $a$.

Ou, plus formellement, si on note $\sigma$ la fonction de $V$ dans $\mathbb{N}$ calculée par l'algorithme à l'étape $i$, que l'on pourrait qualifier de coloration partielle des sommets ($\sigma$ ne sera donc une application qu'après la dernière étape de l'algorithme), on a $DSAT(a) = \begin{cases} d(a) \ si \ \sigma\langle\mathcal{N}(a)\rangle = \emptyset \\ |\sigma\langle\mathcal{N}(a)\rangle| \ sinon \end{cases}$.

L'algorithme est consiste alors à colorer séquentiellement le graphe en prenant à chaque étape un sommet non coloré de degré de saturation maximale, où on impose de prendre un sommet de degré maximal en cas de conflit. Il ne faudra donc pas oublier de mettre à jour à chaque étape le degré de saturation d'un sommet. On remarque également que lorsqu'on colore un sommet $a$, il n'est nécessaire de mettre à jour $DSAT$ que pour les sommets voisins de $a$.

**Programme 3** Algorithme DSATUR

**Entrée** – Un graphe quelconque $G = (V, E)$
**Sortie** – Une coloration propre $c : V \rightarrow \mathbb{N}$

```
1.   n := |V|;
2.   Couleur := Tableau (Taille : n) (Defaut : -1);
3.   Deg := Tableau (Taille : n) (Defaut : 0);
4.   Dsat := Tableau (Taille : n) (Defaut : 0);
5.   # Initialisation #
6.   Pour i de 0 à n-1 faire
7.           Deg.(i) ← d(i);
8.           Dsat.(i) ← d(i)
9.   Fin Pour;
10.  # Boucle principale #
11.  Pour i de 0 à n-1 faire
12.          x := 0;
13.          # Recherche du sommet de Dsat max #
14.          Pour y de 0 à n-1 faire
15.                  Si Couleur.(x) = -1 & Dsat.(y) > Dsat.(x) faire
16.                          x := y
17.                  Fin Si
18.          Fin Pour;
19.          # En cas de conflit, prendre celui de Deg max #
20.          Pour y de 0 à n-1 faire
21.                  Si Couleur.(x) = -1 & Dsat.(y) = Dsat.(x) & Deg.(y) > Deg.(x) faire
22.                          x := y
23.                  Fin Si
24.          Fin Pour;
25.          # Recherche de la plus petite couleur non utilisée dans N(x) #
26.          Libre := Tableau (Taille : ???) (Defaut : true);
27.          Pour y dans N(x) faire
28.                  Si Couleur.(y) ≠ -1 faire
29.                          Libre.(Couleur.(y)) ← false;
30.                  Fin Si
31.          Fin Pour;
32.          index := 0;
33.          Tant Que Libre.(index) = false faire
34.                  index := index + 1
35.          Fin Tant Que;
36.          # Mettre à jour Dsat pour les voisins de x #
37.          Pour y dans N(x) faire
38.                  Si Dsat.(y) = Deg.(y) faire
39.                          Dsat.(y) ← 1;
40.                  Si index ∉ Couleur⟨N(y)⟩ faire
41.                          Dsat.(y) ← Dsat.(y) + 1;
42.                  Fin Si
43.          Fin Pour;
44.          # Affecter la couleur donnée par index à x #
45.          Couleur.(x) ← index
46.  Fin Pour;
47.  Renvoyer Couleur
```

Donnons maintenant quelques explications sur le code proposé : les deux boucles (lignes 14 à 24) pour rechercher le sommet à colorer sont là pour des raisons de clarté, on pourrait très bien se ramener à une seule boucle. Quant à la mise à jour de $DSAT(y)$, pour un voisin $y$ de $x$, et ce avant de colorer $x$, on distingue trois cas :

- $y$ n'a aucun voisin coloré. Alors $DSAT(y) = d(y)$ par définition. Réciproquement l'égalité $DSAT(y) = d(y)$ n'est vraie que dans deux cas : $y$ n'a aucun voisin coloré, ou $y$ a tous ses voisins colorés de couleurs différentes (auquel cas $|\sigma\langle\mathcal{N}(y)\rangle| = |\mathcal{N}(y)| = d(y)$), mais ce n'est pas le cas ici puisque $x \in \mathcal{N}(y)$ et n'est pas encore coloré. En regardant si $DSAT(y) = d(y)$, on est donc sûr qu'il faut régler $DSAT(y)$ à 1.
- $y$ a des voisins colorés, mais aucun de la couleur index : il faut alors incrémenter $DSAT(y)$.
- $y$ a des voisins colorés, dont un au moins de la couleur index : il n'y a alors rien à faire.

La complexité de l'algorithme dépend de l'implémentation que l'on en fait. Notons que si le test de la ligne 40 se fait en temps constant, on obtient alors une complexité temporelle $O(n^2)$. Remarquons qu'il est facile d'effectuer le test de la ligne 40 en temps constant, si on augmente un peu la complexité spatiale de l'algorithme (qui passera de $O(n)$ à $O(n^2)$). En effet supposons que l'on stocke à chaque étape dans un tableau $T$ doublement indexé (une matrice) l'information $T[a,k] = \begin{cases} 1 \; si \; k \in Couleur\langle\mathcal{N}(a)\rangle \\ 0 \; sinon \end{cases}$, alors la condition index $\notin Couleur\langle\mathcal{N}(y)\rangle$ s'écrit $T[y, index] = 0$.

Reste à évaluer le coût temporel de la mise à jour du tableau $T$, qui doit rester $O(n)$ si on ne veut pas augmenter la complexité temporelle totale de l'algorithme. Après avoir affecté la couleur index au sommet $x$, on voit qu'il suffit de mettre à jour $T[y, index]$ pour chaque sommet $y$ voisin de $x$ (car les sommets $y$ pour lesquels $index \in Couleur\langle\mathcal{N}(y)\rangle$ ou encore $x \in \mathcal{N}(y)$ sont exactement les sommets $y \in \mathcal{N}(x)$). On effectue donc l'opération $\forall y \in \mathcal{N}(x), T[y, index] \leftarrow 1$.

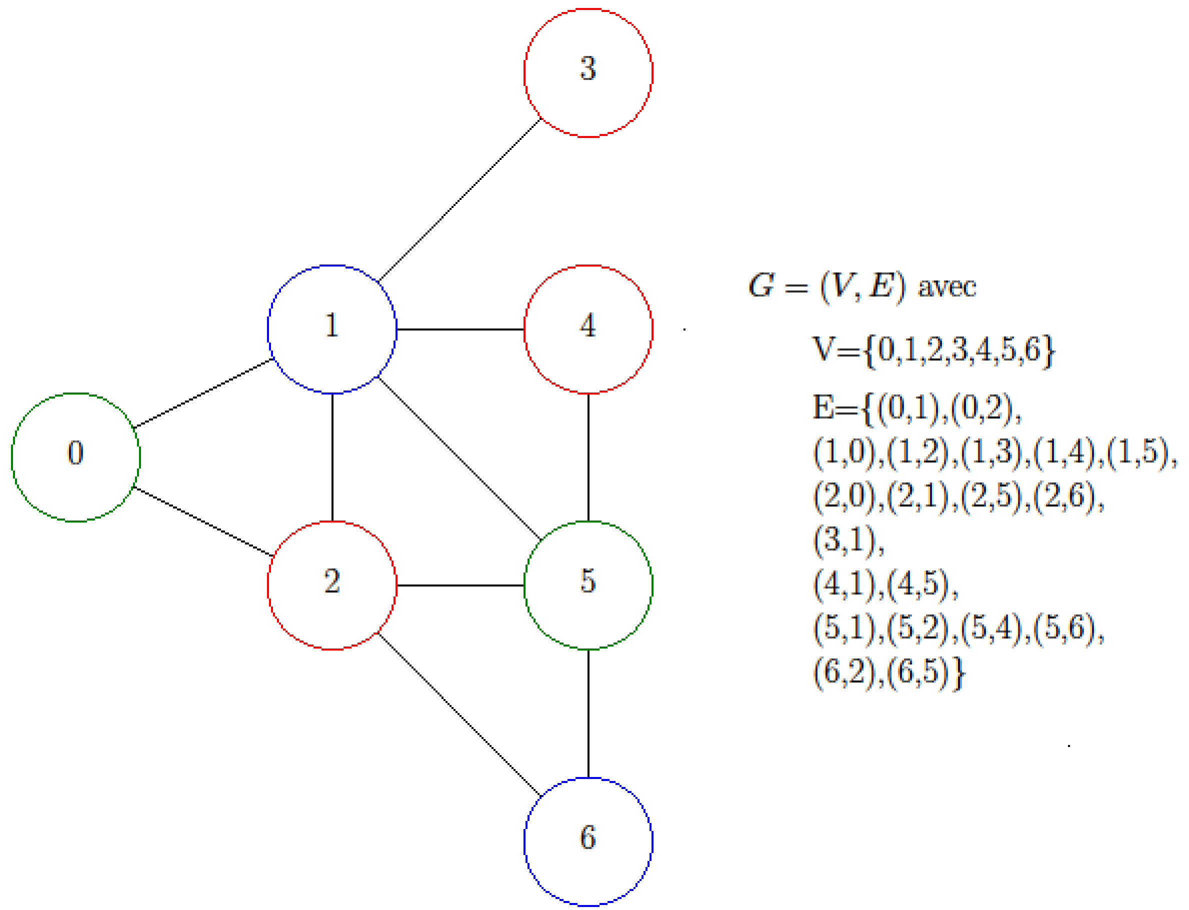



**Fig 2.2** – Exemple d'exécution.

Évolution du tableau DSAT au cours de l'exécution :

| Itération \ DSAT | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| (Initialisation) | 2 | 5 | 4 | 1 | 2 | 4 | 2 |
| 0 | 1 | 5 | 1 | 1 | 1 | 1 | 2 |
| 1 | 1 | 5 | 1 | 1 | 1 | 1 | 2 |
| 2 | 2 | 1 | 1 | 1 | 1 | 2 | 1 |
| 3 | 2 | 2 | 2 | 1 | 2 | 2 | 2 |
| 4 | 2 | 2 | 2 | 1 | 2 | 2 | 2 |
| 5 | 2 | 2 | 2 | 1 | 2 | 2 | 2 |
| 6 | 2 | 2 | 2 | 1 | 2 | 2 | 2 |

# 3 Graphes triangulés : un algorithme linéaire de résolution exact

## 3.1 Généralités sur les graphes triangulés

### 3.1.1 Définitions, propriétés

Après avoir présenté ces quelques algorithmes de résolution approchés, on s'intéresse maintenant à des algorithmes fournissant une réponse optimale exacte. L'inconvénient des heuristiques précédentes est de ne pas fournir systématiquement de coloration propre optimale. Cependant comme il n'existe pas non plus à l'heure actuelle d'algorithme résolvant le problème général en un temps polynomial, on s'intéresse ici à un cas particulier plus facile : celui des graphes triangulés. Commençons par poser quelques définitions :

- Un graphe $G = (V, E)$ est dit *triangulé* si $V$ ne contient pas de trou de longueur supérieure ou égale à 4 : $\{G[S], S \subseteq V\} \cap \{C_k, k \geq 4\} = \emptyset$
- Un graphe $G$ est dit *parfait* si pour tout sous-graphe induit $H$ de $G$, on a $\omega(H) = \chi(H)$.
- Une classe $C$ de graphes est *héréditaire* si pour chaque graphe $G = (V, E) \in C$, pour tout sommet $a$ de $G$, le graphe $G[V \setminus \{a\}]$ est aussi dans $C$ : $\forall G = (V, E) \in C, \forall a \in V, G[V \setminus \{a\}] \in C$
- Un sommet $a$ de $G$ est *simplicial* si son voisinage est une clique, i.e $G[\mathcal{N}(a)]$ est complet.
- Un *schéma* (ou *ordre*) *d'élimination simplicial* est une permutation des sommets du graphe $\sigma : [\![0, n-1]\!] \to V$ telle que pour tout $i$ dans $[\![0, n-1]\!]$, $\sigma(i)$ soit simplicial dans $G[\sigma\langle[\![0, i]\!]\rangle]$.
- Un *ordre parfait* des sommets d'un graphe est un permutation $\sigma : [\![0, n-1]\!] \to V$ telle que pour tout sous-graphe induit $H = G[S]$, une coloration séquentielle de $H$ selon l'ordre induit par la restriction $\sigma_{|S}$ soit une coloration propre optimale de $H$. Si $G$ possède un ordre parfait, on dit que $G$ est *parfaitement ordonnable*.
- Un sous-ensemble de sommets $S \subseteq V$ est un *séparateur* de $G$ si $G[V \setminus S]$ n'est pas connexe. On parle aussi de *$a, b$-séparateur*, où $a$ et $b$ sont deux sommets non connectés de $G[V \setminus S]$. On appelle *séparateur minimal* un séparateur minimal au sens de l'inclusion.

Il est clair d'après sa définition que la classe des graphes parfaits est héréditaire. Les graphes parfaits sont importants dans les problèmes de coloration. En effet Grötschel, Lovász et Schrijver ont démontré en 1984 qu'il était théoriquement possible de colorer un graphe parfait en temps polynomial. Malheureusement leur algorithme, basé sur la méthode dite de l'ellipsoïde, n'a qu'un intérêt théorique car ses performances se révèlent en pratique très mauvaises. De fait, la recherche d'algorithmes pratiques de coloration optimale, que ce soit pour la classe des graphes parfaits ou certaines de ses sous-classes est encore un domaine très actif.

Historiquement, les graphes triangulés sont en fait les premiers graphes dont a démontré la perfection, et on en connait aujourd'hui un algorithme de coloration linéaire, lequel est basé sur une caractérisation des graphes triangulés.

**Proposition 3.1.1.a** : La classe des graphes triangulés est héréditaire.
**Démonstration** : Soit $G = (V, E)$ un graphe triangulé et soit $a \in V$. Notons $H = G[V \setminus \{a\}]$.
On suppose par l'absurde que $H$ n'est pas triangulé : il contient un trou $v_0 \dots v_k$ de longueur $k \geq 4$. Il est alors clair que $v_0 \dots v_k$ est aussi un trou pour $G$ : contradiction avec $G$ triangulé ! ■

On voit ainsi que tout sous-graphe induit d'un graphe triangulé est aussi triangulé.

**Proposition 3.1.1.b** : Un graphe triangulé est parfait.
**Démonstration** : On raisonne par récurrence sur l'ordre du graphe.

- Pour $n = 1$ la propriété est évidente.
- Supposons l'inclusion vérifié pour tout graphe triangulé d'ordre $k < n$, pour un certain $n \geq 2$.

Soit $G = (V, E)$ graphe triangulé d'ordre $n$. La classe des graphes triangulés étant héréditaire, l'hypothèse de récurrence impose donc que tout sous-graphe induit $H$ de $G$, avec $H \neq G$, vérifie $\omega(H) = \chi(H)$. Reste donc à montrer que $\omega(G) = \chi(G)$.

Pour cela, on admet pour l'instant le lemme 2 énoncé ci-après, on considère un sommet simplicial $a \in V$ et on note $H = G[V \setminus \{a\}]$. On a alors $\omega(H) = \chi(H)$. D'après la proposition 1, on sait déjà que $\omega(G) \leq \chi(G)$. Pour démontrer l'autre inégalité, on distingue deux cas :

— $d(a) = \omega(H)$, auquel cas $\omega(G) = \omega(H) + 1$ (car $\{a\} \cup \mathcal{N}(a)$ est une clique de cardinal $\omega(H) + 1$ pour $G$, et $G$ ne peut pas non plus posséder de clique de cardinal $\geq \omega(H) + 2$). Ainsi, en prenant une $\chi(H)$-coloration propre de $H$, il est facile de l'étendre à une $(\chi(H) + 1)$-coloration propre de $G$, d'où $\chi(G) \leq \chi(H) + 1 = \omega(H) + 1 = \omega(G)$. Ce qui démontre bien l'égalité attendue.

— $d(a) \neq \omega(H)$. Alors a fortiori $d(a) < \omega(H)$, car $\mathcal{N}(a)$ est une clique de $H$. On considère alors une $\chi(H)$-coloration propre de $\sigma : V \setminus \{a\} \to [\![0, \chi(H) - 1]\!]$. On voit que $\sigma\langle\mathcal{N}(a)\rangle$ est inclus strictement dans $[\![0, \chi(H) - 1]\!]$ (car $|\sigma\langle\mathcal{N}(a)\rangle| \leq |\mathcal{N}(a)| = d(a) < \omega(H) = \chi(H)$). On peut donc prolonger $\sigma$ en une $\chi(H)$-coloration propre de $G$ $\tilde{\sigma} : V \to [\![0, \chi(H) - 1]\!]$ en prenant pour $a$ $\tilde{\sigma}(a) \in [\![0, \chi(H) - 1]\!] \setminus \sigma\langle\mathcal{N}(a)\rangle$. Finalement on a bien montré que $\chi(G) \leq \chi(H) = \omega(H) \leq \omega(G)$ ■

### 3.1.2 Une caractérisation des graphes triangulés

Notre algorithme de coloration pour les graphes triangulés n'utilise pas le fait que ceux-ci sont parfaits. Il repose en fait sur une caractérisation des graphes triangulés, établie par Fulkerson et Gross en 1965, qui établit l'équivalence entre "$G$ est un graphe triangulé" et "$G$ possède un schéma d'élimination simplicial". Afin de démontrer cette équivalence, on utilise les lemmes suivants :

**Lemme 1** : Soit $G = (V, E)$ un graphe triangulé. Alors tout séparateur minimal $S \subseteq V$ est une clique.

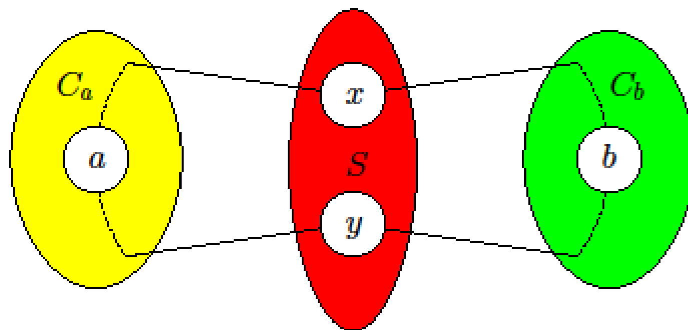



Fig 3.1

**Démonstration** : On suppose l'existence de $S \subseteq V$ séparateur minimal, et on considère deux sommets $a, b$ non connectés dans $G[V \setminus S]$. Notons $C_a$ (respectivement $C_b$) la composante connexe de $G[V \setminus S]$ contenant $a$ (respectivement $b$).

Soient $x, y \in S$. Supposons par l'absurde $x, y$ non-adjacents.

Alors $x$ est adjacent à un sommet de $C_a$. En effet sinon $x$ et $a$ non connectés dans $G[V \setminus S']$, où $S' = S \setminus \{x\}$, et donc contradiction avec $S$ minimal. De même $x$ est adjacent à un sommet de $C_b$, et $y$ est adjacent à un sommet de $C_a$ et à un sommet de $C_b$. On considère alors un cycle de $G$ de longueur minimale parmi ceux dont les sommets sont dans $C_a \cup C_b \cup \{x, y\}$ et qui passent par $x$, par $y$, par un sommet de $C_a$ et par un sommet de $C_b$ (existe d'après ce qui précède). Un tel cycle est alors de longueur supérieure à 4.

Or un tel cycle n'a pas de corde ; en effet, il est composé d'une chaîne de $x$ à $y$ qui est un chemin sans corde (par minimalité de la taille du cycle), et d'une chaîne de $y$ à $x$ qui est de même un chemin sans corde. De plus il n'y a pas de corde entre un sommet du cycle appartenant à $C_a$ et un autre appartenant à $C_b$ (composantes connexes disjointes), et il n'y a pas de corde entre $x$ et $y$ par hypothèse. Finalement on a exhibé un trou de longueur supérieure à 4 : contradiction avec $G$ triangulé.

Ainsi $x$ et $y$ sont adjacents et $S$ est une clique ■

**Lemme 2** : Soit $G$ un graphe triangulé. Alors $G$ admet au moins un sommet simplicial.
**Démonstration** : On démontre par récurrence sur l'ordre de $G$ la propriété suivante : pour $G$ graphe triangulé, soit $G$ est complet (et donc tous ses sommets sont simpliciaux), soit il admet au moins deux sommets simpliciaux non adjacents.

- Pour $n = 1$ c'est évident.
- Soit un $G = (V, E)$ un graphe triangulé d'ordre $n > 1$. Supposons la propriété établie pour tout graphe triangulé d'ordre strictement inférieur à $n$. On suppose alors $G$ non complet.

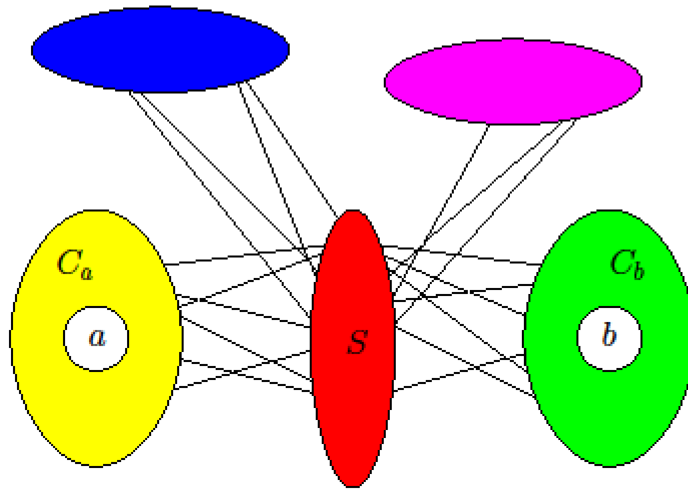



Fig 3.2

Alors il existe deux sommets de $G$ non adjacents, et $G$ possède des ensembles séparateurs. On considère alors un séparateur minimal $S$. Il vient que $G[V \setminus S]$ possède deux sommets $a$ et $b$ appartenant à des composantes connexes respectives $C_a$ et $C_b$ distinctes. On distingue alors plusieurs cas :

— $C_a \cup S$ est une clique. Alors, comme $\mathcal{N}(a) \subseteq (C_a \cup S)$ (aucun sommet d'une autre composante connexe de $G[V \setminus S]$ n'est adjacent à $a$), il vient que $\mathcal{N}(a)$ est une clique est que $a$ est simplicial (dans $G$).

— Comme $|C_a \cup S| < n$ ($b \notin (C_a \cup S)$), on applique l'hypothèque de récurrence à $G[C_a \cup S]$, qui n'est pas complet : il existe deux sommets non adjacents $x, y \in (C_a \cup S)$ qui vérifient que $\mathcal{N}(x) \cap (C_a \cup S)$ est une clique ($x$ simplicial dans $G[C_a \cup S]$), et de même pour $y$. On distingue à nouveau plusieurs cas :

❖ $x \in C_a$. Alors comme précédemment $\mathcal{N}(x) \subseteq (C_a \cup S)$, et donc $\mathcal{N}(x) = \mathcal{N}(x) \cap (C_a \cup S)$ est une clique, et $x$ simplicial (dans $G$).

❖ $y \in C_a$. Alors de même $y$ simplicial (dans $G$).

❖ Sinon $x, y \in S$. On a alors deux sommets non adjacents dans un séparateur minimal, contradiction avec $S$ est une clique (cf. Lemme 1) !

Finalement on a exhibé un sommet simplicial appartenant à $C_a$. On procède de même pour $C_b$ afin de montrer l'existence d'un deuxième sommet simplicial de $G$ non adjacent au premier ∎

Ce dernier lemme conduit à l'établissement de la caractérisation annoncée précédemment :
**Proposition 3.1.2** :

Un graphe est triangulé si et seulement si il admet un schéma d'élimination simplicial.

**Démonstration** :

⇒ On applique récursivement le résultat du Lemme 2 précédent :
On considère un graphe triangulé $G = (V, E)$. Il admet un sommet simplicial $a$. Alors d'après la proposition 6, $G[V \setminus \{a\}]$ est aussi triangulé. On peut donc répéter l'opération jusqu'à arriver au graphe $(\emptyset, \emptyset)$.

⇐ On raisonne par contraposée. Si un graphe $G$ n'est pas triangulé, alors il possède un trou de longueur supérieure ou égale à $4$. Or aucun des sommets d'un tel cycle ne peut être "éliminé

simplicialement" : le premier sommet à éliminer aura forcément dans son voisinage dans le sous-graphe induit par les sommets restant deux sommets non adjacents (ses voisins dans le cycle) ■

#### 3.1.3 Coloration séquentielle

La présence d'un tel schéma d'élimination dans un graphe triangulé va être utile par la suite, car en en appelant l'algorithme de coloration séquentielle sur $G$ avec ce schéma d'élimination simplicial, on obtient une coloration propre optimale :

**Proposition 3.1.3.a** : Soit $G = (V, E)$ graphe et $\sigma : [\![0, n-1]\!] \to V$ schéma d'élimination simplicial. Alors, l'appel :

Coloration séquentielle **(**Graphe : $G$**)** **(**Numérotation : $\sigma$**)**

Fournit une coloration propre optimale de $G$.

**Démonstration** : Il s'agit de montrer que la coloration séquentielle obtenue pour le $G$ est bien une $\omega(G)$-coloration (étant donné que $\omega(G) = \chi(G)$ : le graphe est parfait). En remarquant que $\sigma\Big|_{[\![0, n-2]\!]}$ est toujours un schéma d'élimination simplicial pour $G' = G[V \setminus \{\sigma(n-1)\}]$, et que $G'$ reste triangulé par hérédité, on raisonne alors par récurrence sur l'ordre du graphe.

- Pour $n = 1$, la propriété est triviale.
- Soit $n > 1$. Supposons que pour tout graphe d'ordre $n - 1$, l'appel Coloration séquentielle avec un schéma d'élimination simplicial fournisse une coloration propre optimale.

On se donne alors $G$ graphe d'ordre $n$ et $\sigma : [\![0, n-1]\!] \to V$ schéma d'élimination simplicial. Notons $x = \sigma(n-1)$, $G' = G[V \setminus \{x\}]$ ainsi que $c : V \to \mathbb{N}$ la coloration propre fournie par l'algorithme. Par hypothèse de récurrence, on a donc $c\langle V \setminus \{x\}\rangle = [\![0, \omega(G') - 1]\!]$. Deux cas se présentent alors :

❖ 1er cas : $\omega(G) = \omega(G') + 1$

Alors il est clair que $|\mathcal{N}(x)| = \omega(G')$, et que tous les voisins de $x$ sont colorés de couleurs différentes. Ainsi le sommet $x$ va être coloré d'une autre couleur, et on aura donc $c(x) = \omega(G')$.

❖ 2ème cas : $\omega(G) = \omega(G')$

Alors on a $|\mathcal{N}(x)| < \omega(G')$, et donc $\exists p \in [\![0, \omega(G') - 1]\!], p \notin c\langle\mathcal{N}(x)\rangle$. L'algorithme va donc colorer $x$ avec une couleur $q \leq p$.

Dans tous les cas, on obtient bien que $|c(V)| = \omega(G) = \chi(G)$ ■

Il reste alors à trouver un tel ordre en un temps raisonnable dans le cas d'un graphe triangulé : c'est le but de l'algorithme de parcours en largeur lexicographique. Cet algorithme va également permettre de reconnaître un graphe triangulé, car il produit systématiquement un schéma d'élimination simplicial dans le cas où $G$ est triangulé (et donc s'il n'en produit pas, $G$ n'est pas triangulé).

Notons également la :

**Proposition 3.1.3.b** : Tout graphe triangulé est parfaitement ordonnable. Et un schéma d'élimination simplicial est un ordre parfait.

Démonstration : On considère un graphe triangulé $G = (V, E)$, ainsi qu'un schéma d'élimination simplicial pour $G$ $\sigma : [\![0, n-1]\!] \to V$. On vérifie alors facilement que pour tout sous graphe induit $H = G[S]$, $\sigma_{|S}$ induit bien un schéma d'élimination simplicial pour $H$.

### 3.2 Rappel : Parcours en largeur – BFS.

Avant de présenter l'algorithme de parcours en largeur lexicographique, dit Lex_BFS, il convient de rappeler un premier algorithme, plus classique, dit de parcours en largeur, duquel s'inspire Lex_BFS.

Le parcours en largeur (ou BFS, pour "Breadth First Search") consiste à visiter les différents sommets du graphe selon un certain ordre. Un tel ordre sur les sommets est aussi appelé "ordre BFS". On marque les sommets visités, et on enfile dans une liste d'attente les voisins non encore marqués du sommet courant, afin de les visiter plus tard. Cela donne l'algorithme suivant :

**Programme 4** Parcours en Largeur
**Entrée** – Un graphe connexe $G = (V, E)$, un sommet source $s \in V$
**Sortie** – Une bijection $\sigma : V \to [\![0, n-1]\!]$

```
1    n := |V|;
2    Marque := Tableau (Taille : n) (Defaut : false);
3    Numero := Tableau (Taille : n) (Defaut : -1);
4    compteur := 0;
5    File := File_Vide;
6    Marque.(x) ← true;
7    File.ajouter_fin(s);
8    Tant Que File ≠ File_Vide faire
9        x := File.retirer_tête();
10       Numero.(x) ← compteur;
11       compteur := compteur + 1;
12       Pour y dans N(x) faire
13           Si Marque.(y) = false faire
14               Marque.(x) ← true;
15               File.ajouter_fin(y)
16           Fin Si
17       Fin Pour
18   Fin Tant Que;
19   Renvoyer Numero
```

L'intérêt par rapport à un algorithme de parcours en profondeur par exemple, est que cet algorithme explore les sommets selon un ordre différent : il explore les sommets par "distance" croissante à la source. Si le graphe est connexe, il va alors parcourir tous les sommets du graphe. Plus formellement, on définit la distance entre deux sommets $x$ et $y$ du graphe comme étant le minimum des longueurs des chemins reliant $x$ à $y$. On notera $d(x, y)$ cette distance :

$$d(x, y) = \min\{k \in \mathbb{N}, \exists v_0, \dots, v_k \in V, (v_i)_{0 \le i \le k} \; chemin, v_0 = x, v_k = y\}$$

Vérifions qu'il s'agit bien d'une distance au sens mathématique du terme (on munit alors l'ensemble $V$ des sommets d'une structure d'espace métrique) :

— **Positivité** : $d : V \times V \to \mathbb{R}^+$ (le graphe est connexe)

— **Symétrie** : $\forall x, y \in V, d(x, y) = d(y, x)$

Soient $x, y \in V$ et $v_0 \dots v_k$ chemin reliant $x$ à $y$, où $k = d(x, y)$. Alors il est clair que $v_k \dots v_0$ est un chemin reliant $y$ à $x$ de longueur $k$ (le graphe étant symétrique, les arcs sont des arêtes).
Ainsi on a $d(y, x) \le d(x, y)$. On montre de même la réciproque, d'où la symétrie.

— **Inégalité triangulaire** : $\forall x, y, z \in V, d(x, z) \le d(x, y) + d(y, z)$

Soient $x, y, z \in V$, chemin reliant $x$ à $y$ et $v_p \dots v_{p+q}$ chemin reliant $y$ à $z$, où $p = d(x, y)$ et $q = d(y, z)$.
Alors $v_0 \dots v_{p+q}$ est un chemin reliant $x$ à $z$, d'où $d(x, z) \le p + q = d(x, y) + d(y, z)$

— **Condition de séparation** : $\forall x, y \in V, (d(x, y) = 0) \Leftrightarrow (x = y)$

Soient $x, y \in V$. On suppose $d(x, y) = 0$. Alors on a un chemin de longueur nulle $v_0$ reliant $x$ à $y$, d'où $x = v_0 = y$. Réciproquement si $x = y$ il est clair que $v_0 = x$ est un chemin de longueur nulle reliant $x$ à $y$, d'où $d(x, y) = 0$.

On montre alors les propriétés suivantes sur le programme 4 :

**Proposition 3.2** : La numérotation obtenue est une application $\sigma : V \to [\![0, n-1]\!] \cup \{-1\}$ qui vérifie :

**a.** $\sigma(x) \neq -1$ si et seulement si $x$ est accessible depuis $s$ (donc $x$ et $s$ sont dans la même composante connexe vu que le graphe est supposé non-orienté). Il en découle immédiatement que comme $G$ est connexe, $\sigma$ est une bijection de $V$ dans $[\![0, n-1]\!]$.

**b.** La complexité de l'algorithme est $O(n + m)$

**c.** Les sommets sont visités par ordre croissant de distance à la source $s$ :

$$\forall x, y \in V, \big(\sigma(x) \le \sigma(y)\big) \Rightarrow \big(d(s, x) \le d(s, y)\big)$$

**Démonstration** :

**a.** Notons $S$ l'ensemble des descendants de $s$ (les sommets accessibles depuis $s$), et $T$ l'ensemble des sommets traités – pour lesquels $(\sigma(x) \neq -1)$ – à une étape donnée de l'algorithme. On note de même $F$ l'ensemble des sommets enfilés – qui ne sont pas encore dans $T$ mais pour lesquels Marque.(x) = *true* – à une étape donnée. On montre par induction que les sommets marqués sont des descendants de $s$ en vérifiant les invariants :

- $T \subseteq S$
- $\forall y \in F, \exists x \in T, (x, y) \in E$

L'initialisation vient du fait que le premier sommet marqué (et donc ensuite dépilé) est $s \in S$.

Réciproquement, à la fin de l'exécution de l'algorithme on a bien $S \subseteq T$. En effet supposons par l'absurde qu'il existe un sommet non marqué. On peut alors considérer un chemin $v_0 \dots v_k$ avec $x = v_{k-1} \in T$ et $y = v_k \notin T$. Mais alors à la fin de l'étape où $x$ est dépilé, on aura $y \in F$. Or tout sommet dans $F$ va être dépilé plus tard, donc on aura à la fin de l'algorithme $y \in T$ : contradiction ■

**b.** L'algorithme se termine car tout sommet est enfilé au plus une fois (un sommet $x$ pour lequel Marque.(x) = *true* ne sera pas enfilé). Ainsi la boucle principale (lignes 8 à 18) s'exécute au plus $|V| = n$ fois. La boucle intérieure (lignes 12 à 17) s'exécute alors au plus $\sum_{x \in V} |\mathcal{N}(x)| = m$ fois. D'où une complexité totale $O(n + m)$ ■

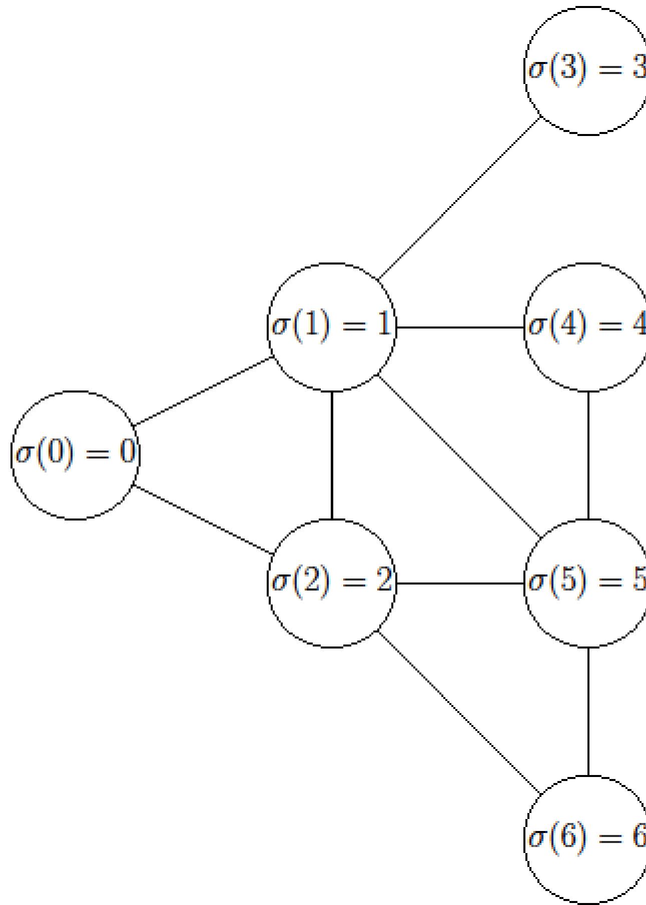
**c.** Notons $S_k = \{x \in V, d(s, x) = k\}$ et $N = \max\{k \in \mathbb{N}, S_k \neq \emptyset\}$ (existe car il n'y aucun chemin de longueur strictement supérieure à $n - 1$). On montre alors par récurrence sur $k$ que $\forall k \in [\![0, N]\!]$ il existe une étape de l'algorithme pour laquelle $T = \bigcup_{0 \leq i \leq k-1} S_i$ et $F = S_k$ :

- Pour $k = 0$ : Au début de l'algorithme, $T = \emptyset$ et $F = S_0 = \{s\}$
- Supposons maintenant la propriété vraie pour un certain $k \in [\![0, N - 1]\!]$.

Il existe alors une étape $p$ de l'algorithme pour laquelle laquelle $T = \bigcup_{0 \leq i \leq k-1} S_i$ et $F = S_k$. Ainsi, à l'étape $q = p + |S_k|$, on aura bien $T = \bigcup_{0 \leq i \leq k} S_i$. Reste à prouver que les sommets enfilés entre les étapes $p$ et $q$ sont exactement ceux de $S_{k+1}$. Cela vient du fait que tout sommet de $S_{k+1}$ est adjacent à un sommet de $S_k$ (ils vont donc être enfilés, et aucun ne sera dépilé avant l'étape $q$) ; et réciproquement si un sommet $x$ est adjacent à un sommet de $S_k$, alors soit il appartient à un ensemble $S_j$ avec $j > k$, et dans ce cas $j = k + 1$, soit il appartient à un ensemble $S_j$ avec $j \leq k$, et dans ce cas il ne sera pas empilé (car vérifie Marque.(x) = *true* : il appartient donc déjà à $T$ ou à $F$).

Cela démontre la condition souhaitée sur l'ordre de parcours, car lors d'une étape de l'algorithme, en notant $\sigma$ la numérotation finale obtenue, on a bien $\forall x \in T, \forall y \in F, \sigma(x) < \sigma(y)$. D'où le résultat par contraposée ■

**Fig 3.3** – Résultat d'un parcours BFS sur un exemple : 0123456




Cependant, il s'avère que les conditions imposées sur l'ordre de parcours avec un tel algorithme ne sont pas assez fortes pour obtenir une condition satisfaisante sur l'ordre obtenu. Afin de préciser l'ordre selon lequel les sommets vont être visités au sein d'un même ensemble $S_k$, on modifie l'algorithme précédent de telle sorte que "les sommets ayant le plus de voisins déjà visités soient visités en premier".

Avant de détailler cet algorithme en section 3.4, on rappelle quelques propriétés utiles sur les mots, et notamment sur la notion d'ordre lexicographique.

### 3.3 Mots et Ordre Lexicographique

On considère l'*alphabet* $A = [\![1, n]\!]$, ensemble fini non vide (ses éléments sont appelés des *lettres*), que l'on muni de la relation d'ordre habituelle sur les entiers, notée $\leq$. Un *mot* sur l'alphabet $A$ est une suite finie d'éléments de $A$ de la forme $x_0 \dots x_{p-1}$. On note $A^*$ l'ensemble des mots sur $A$. On note $\varepsilon = ()$ le *mot vide*. La *longueur* d'un mot $u \in A^*$ sera notée $|u|$ : si $u = x_0 \dots x_{p-1} \in A^*$, alors $|u| = p$ ; et par convention $|\varepsilon| = 0$.

Si $u = x_0 \dots x_{p-1}$ et $i \in [\![0, p-1]\!]$, on note $u[i] = x_i$, la $i$-ième lettre de $u$. Pour deux mots $u = x_0 \dots x_{p-1}$ et $v = y_0 \dots y_{q-1}$, on note $u \cdot v = uv = x_0 \dots x_{p-1} y_0 \dots y_{q-1}$ la *concaténation* de $u$ et $v$. Cette opération est associative et admet $\varepsilon$ comme élément neutre (autrement dit, $(A^*, \cdot)$ est un monoïde). Un mot $v$ est *préfixe* d'un autre mot $u$ s'il existe un mot $w$ tel que $u = vw$. On note $Pre(u)$ l'ensemble des préfixes de $u$.

À partir de la relation d'ordre stricte déduite de $\leq$, on définit différentes relations binaires sur $A^*$ :

- Pour deux mots $u$ et $v$, on note $u \lhd v$ s'il existe $i \in \mathbb{N}$ vérifiant $i < \min(|u|, |v|)$, $u[i] < v[i]$ et $\forall j \in [\![0, i-1]\!], u[j] = v[j]$.
- On note ensuite $u \prec v$ si $u \in Pre(v) \setminus \{v\}$ ou $u \lhd v$.
- Enfin, $u \preccurlyeq v$ lorsque $u \prec v$ ou $u = v$.

**Proposition 3.3** : On démontre alors les propriétés suivantes sur $\lhd$, $\prec$ et $\preccurlyeq$ :

- **a.** $\lhd$ est une relation d'ordre stricte, non totale, compatible avec $\cdot$
- **b.** $\prec$ est une relation d'ordre stricte totale, compatible avec $\cdot$
- **c.** $\preccurlyeq$ est une relation d'ordre totale, compatible avec $\cdot$

**Démonstration** :

**a.** On vérifie les différents axiomes :

— **Irréflexivité** : $\forall u \in A^*, u \ntriangleleft u$

Soit $u \in A^*$. Supposons par l'absurde $u \lhd u$. Alors $\exists i < |u|, u[i] < u[i]$, ce qui est impossible car $<$ est une relation d'odre stricte (et donc irréflexive).

— **Transitivité** : $\forall u, v, w \in A^*, (u \lhd v \; et \; v \lhd w) \Rightarrow (u \lhd w)$

Soient $u, v, w \in A^*$ tels que $u \lhd v$ et $v \lhd w$.
Soit $i_0 < \min(|u|, |v|)$ tel que $u[i_0] < v[i_0]$ et $\forall j < i_0, u[j] = v[j]$.
Soit $i_1 < \min(|v|, |w|)$ tel que $v[i_1] < w[i_1]$ et $\forall j < i_1, v[j] = w[j]$.
On pose $i = \min(i_0, i_1)$. On a bien $i < \min(|u|, |w|)$, $u[i] < w[i]$ et $\forall j < i, u[j] = v[j]$. D'où $u \lhd w$

— **Non totale** : $\exists u, v \in A^*, (u \ntriangleleft v \; et \; u \neq v \; et \; v \ntriangleleft u)$

On prend par exemple $u = \varepsilon$ et $v = a \in A$

— **Compatible** avec $\cdot$ : $\forall u, v, w \in A^*, u \lhd v \Rightarrow (uw \lhd vw \; et \; wu \lhd wv)$

Découle trivialement des définitions.
Mieux : on a même $\forall u, v, w, w' \in A^*, u \lhd v \Rightarrow uw \lhd vw'$ ■

**b.** De même :

— **Irréflexivité** : $\forall u \in A^*, u \nprec u$

Soit $u \in A^*$. Supposons par l'absurde $u \prec u$. Alors, d'après l'irréflexivité de $\lhd$, on a $u \in Pre(u) \setminus \{u\}$, ce qui est impossible.

— **Transitivité** : $\forall u, v, w \in A^*, (u \prec v \; et \; v \prec w) \Rightarrow (u \prec w)$

Soient $u, v, w \in A^*$ tels que $u \prec v$ et $v \prec w$. Quatre cas se présentent alors :

❖ <u>1er cas</u> : $u \in Pre(v) \setminus \{v\}$ et $v \in Pre(w) \setminus \{w\}$

Alors on a $v = uv'$ et $w = vw'$, avec $v', w' \in A^* \setminus \{\varepsilon\}$. D'où $w = uv'w'$, avec $v'w' \in A^* \setminus \{\varepsilon\}$.
Ainsi on a $u \in Pre(w) \setminus \{w\}$ et $u \prec w$.

❖ <u>2ème cas</u> : $u \lhd v$ et $v \lhd w$

Alors, par transitivité de $\lhd$, on a encore $u \prec w$.

❖ <u>3ème cas</u> : $u \in Pre(v) \setminus \{v\}$ et $v \lhd w$

On écrit $v = uv'$, avec $v' \in A^* \setminus \{\varepsilon\}$.
Soit aussi $i < \min(|v|, |w|)$ tel que $v[i] < w[i]$ et $\forall j < i, v[j] = w[j]$.
Alors, si $i < |u|$, on a bien $u \lhd w$. Sinon il vient que $u \in Pre(w) \setminus \{w\}$.
Finalement on a toujours $u \prec w$.

❖ <u>4ème cas</u> : $u \lhd v$ et $v \in Pre(w) \setminus \{w\}$

Soit $i < \min(|u|, |v|)$ tel que $u[i] < v[i]$ et $\forall j < i, u[j] = v[j]$.
On écrit aussi $w = vw'$, avec $w' \in A^* \setminus \{\varepsilon\}$.
Alors, comme $i < |w|$, on en déduit facilement que $u \lhd w$.

— **Totale** : $\forall u, v \in A^*, (u \prec v \; ou \; u = v \; ou \; v \prec u)$

Soient $u, v \in A^*$. Plusieurs cas se présentent :

❖ <u>1er cas</u> : $u \lhd v$ ou $v \lhd u$

❖ <u>2ème cas</u> : $u \ntriangleleft v$ et $v \ntriangleleft u$

Alors, $\forall j < \min(|u|, |v|), u[j] = v[j]$. De là on déduit que :
Si $|u| = |v|, u = v$.
Si $|u| < |v|, u \prec v$.
Si $|v| < |u|, v \prec u$.

— **Compatible** avec $\cdot$ : $\forall u, v, w \in A^*, u \prec v \Rightarrow (uw \prec vw \; et \; wu \prec wv)$

Vrai si $u \lhd v$. C'est encore évident si $u \in Pre(v) \setminus \{v\}$ ■

**c.** On vérifie facilement :
— **Réflexivité** : $\forall u \in A^*, u \preccurlyeq u$
— **Transitivité** : $\forall u, v, w \in A^*, (u \preccurlyeq v \; et \; v \preccurlyeq w) \Rightarrow (u \preccurlyeq w)$
Découle immédiatement de la transitivité de $\prec$
— **Antisymétrie** : $\forall u, v \in A^*, (u \preccurlyeq v \; et \; v \preccurlyeq u) \Rightarrow (u = v)$
Découle des propriétés de $\prec$ :
Par contraposée, la transitivité nous donne que $\forall u, v, w \in A^*, (u \nprec w) \Rightarrow (u \nprec v \; ou \; v \nprec w)$
Appliquée à $u, v \in A^*$ (on prend $w = u$), et de l'irréflexivité de $\prec$, on déduit que $(u \prec v \; et \; v \prec u)$ impossible. De fait, si $u \preccurlyeq v$ et $v \preccurlyeq u$, il vient que $u = v$.
— **Totale** : $\forall u, v \in A^*, (u \preccurlyeq v \; ou \; v \preccurlyeq u)$
— **Compatible** avec $\cdot$ : $\forall u, v, w \in A^*, u \preccurlyeq v \Rightarrow (uw \preccurlyeq vw \; et \; wu \preccurlyeq wv)$ ■

### 3.4 Parcours en largeur lexicographique – Lex_BFS

On se donne alors un graphe $G = (V, E)$, où $V = [\![1, n]\!]$. On note toujours $A = [\![1, n]\!]$, ou encore $A = \{a_1, \dots, a_n\}$, avec $a_i = i$ pour $1 \leq i \leq n$. On associe alors à chaque sommet $x \in V$ une étiquette, notée $L(x) \in A^*$, un mot sur $A$, initialement égal à $\varepsilon$. On va donc parcourir les sommets du graphe en effectuant les opérations suivantes :
Itération $i \in [\![0, n-1]\!]$ :

- Prendre un sommet $x$ d'étiquette maximale pour $\preccurlyeq$ parmi les sommets non visités.
- Pour chaque voisin $y \in \mathcal{N}(x)$, effectuer la mise à jour $L(y) \leftarrow L(y) \cdot a_{n-i}$

Cela nous donne l'algorithme suivant, dit de parcours en largeur lexicographique (ou Lex_BFS, pour "Lexicographic Breadth First Search") :

**Programme 5** Parcours en Largeur Lexicographique
**Entrée** – Un graphe quelconque $G = (V, E)$
**Sortie** – Une bijection $\sigma : V \rightarrow [\![0, n-1]\!]$

```
1     n := |V|;
2     Numero := Tableau (Taille : n) (Defaut : -1);
3     Label := Tableau (Taille : n) (Defaut : "");
4     Pour i de 0 à n-1 faire
5           Soit x tel que Label.(x) = max{Label.(y), Numero.(y)=-1};
6           Numero.(x) ← i;
7           Pour y dans N(x) faire
8                 Si Numero.(y) = -1 faire
9                       Label.(y) ← Label.(y) ^ String_of_Int (n-i)
10                Fin Si
11          Fin Pour
12    Fin Pour;
13    Renvoyer Numero
```

**NB** : Un mot sur $A$ est représenté par une chaîne de caractère. Le mot vide $\varepsilon$ est représenté par la chaîne vide "", et la concaténation $\cdot$ est notée ^.

Tous les sommets sont bien sûr visités. La question de la complexité sera abordée dans le paragraphe suivant. L'ordre obtenu est aussi qualifié d'"ordre lexicographique", ou "ordre Lex_BFS". Le lien avec le parcours en largeur tient à ce que la numérotation obtenue est aussi "croissante" pour la distance à la source $s$, si $s = \sigma^{-1}(0)$. Ainsi une numérotation obtenue par Lex_BFS est compatible avec la proposition 3.2.c relative au parcours en largeur : un ordre lexicographique est donc un ordre BFS. En ce sens Lex_BFS est en quelque sorte une "particularisation" de BFS, car il impose également des contraintes supplémentaires, qui permettent d'obtenir un schéma d'élimination simplicial dans le cas où $G$ est triangulé.

**Proposition 3.4.a** : Les sommets visités par Lex_BFS sont de distance croissante à la source $s$ :

$$\forall x, y \in V, (\sigma(x) \leq \sigma(y)) \Rightarrow (d(s,x) \leq d(s,y))$$

**Démonstration** : Notons $T$ l'ensemble des sommets visités (pour lesquels Numero.(x) $\neq 0$), et $X$ l'ensemble des sommets non visités d'étiquette $\neq \varepsilon$, à une étape donnée de l'algorithme. On reprend alors les notations $S_k = \{x \in V, d(s,x) = k\}$ et $N = \max\{k \in \mathbb{N}, S_k \neq \emptyset\}$. On montre ensuite par récurrence sur $k$ que $\forall k \in [\![0, N]\!]$ il existe une étape de l'algorithme pour laquelle $T = \bigcup_{0 \leq i \leq k-1} S_i$ et $X = S_k$ :

- Pour $k = 0$ : Au début de l'algorithme, $T = \emptyset$ et $X = S_0 = \{s\}$ (avant la boucle).
- Supposons maintenant la propriété vraie pour un certain $k \in [\![0, N-1]\!]$.

Il existe alors une étape $p$ de l'algorithme pour laquelle laquelle $T = \bigcup_{0 \leq i \leq k-1} S_i$ et $X = S_k$. Ainsi tout étiquette de $X$ commence par une lettre $a_j$, avec $j \geq n - p$. On voit donc qu'entre les étapes $p$ et $q = p + |S_k|$, on aura toujours $L(y) \prec L(x)$ si $x \in S_k$ et $y \in \bigcup_{k+1 \leq i \leq N} S_i$ : si $y \in X \cap \bigcup_{k+1 \leq i \leq N} S_i$, on aura $L(y) \lhd L(x)$ ; et si $y \in (\bigcup_{k+1 \leq i \leq N} S_i) \setminus X$, $L(y) = \varepsilon$ et le résultat est encore vrai. De fait, les sommets visités entre les étapes $p$ et $q$ sont exactement ceux de $S_k$. Or tout sommet ajouté à $X$ en visitant un sommet de $S_k$ appartiendra forcément à $S_{k+1}$ (il ne peut pas appartenir à $T$ ni à $X$ !). Réciproquement tout sommet $y$ de $S_{k+1}$ ayant un voisin dans $S_k$, ce dernier sera visité entre les étapes $p$ et $q$, donc $y$ appartiendra à $X$ à l'étape $q$ ■

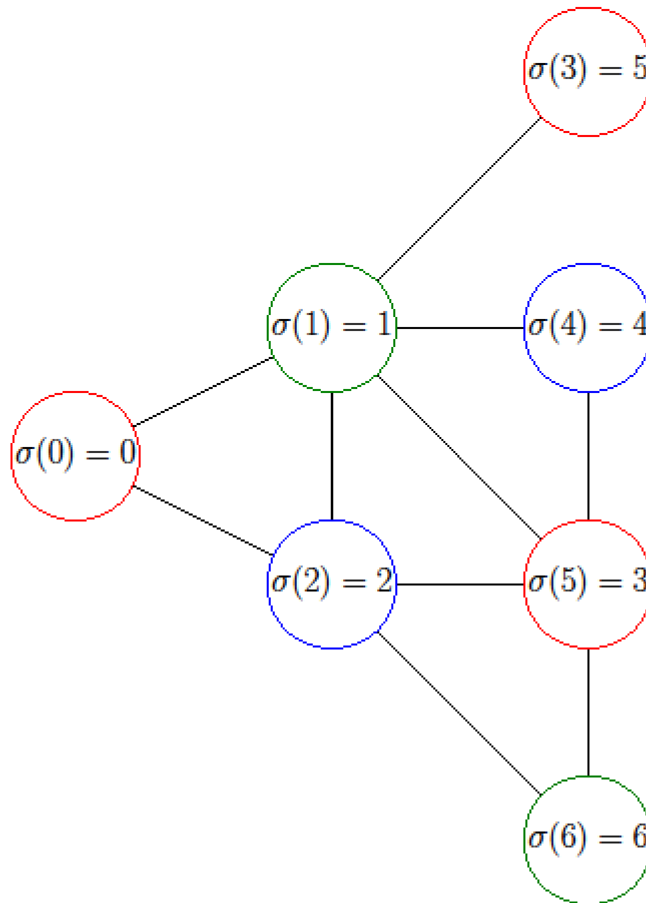



**Fig 3.4** – Résultat d'un parcours LexBFS et coloration obtenue sur un exemple : 0125436

À propos de la **Fig 3.4**, on notera que l'ordre obtenu avec BFS sur ce même graphe ne satisfait pas les spécifications de LexBFS. En effet si les trois premiers sommets visités sont respectivement 0,1,2, en regardant les étiquettes associées on voit que le sommet suivant devant être visité est 5, et non 3.

En pratique le choix du sommet source $s$ importe peu. On se contente donc de numéroter les étiquettes sur l'alphabet $A' = [\![1, n]\!]$. La propriété importante de cet algorithme est la suivante :

**Proposition 3.4.b** : Soit $G = (V, E)$ graphe triangulé et $\sigma : V \to [\![0, n-1]\!]$ ordre lexicographique pour $G$. Alors $\sigma^{-1}(n-1)$ est simplicial dans $G$.

**Démonstration** : On note $L_k(x)$ l'étiquette du sommet $x$ avant l'itération $k$ de l'algorithme. On a donc $L_0(x) = \varepsilon$ pour $x \neq s$, et $L_0(s) = a_n$. Notons aussi $X_k = \{x, \sigma(x) \leq k\}$. Si $G$ est connexe les $X_k$ induisent tous des sous-graphes connexes. On suppose alors sans perte de généralité que $G$ est connexe. On commence ensuite par établir un premier lemme :

**Lemme** : $\forall x, y \in V, \forall k \in [\![0, n-1]\!], \big(L_k(x) \prec L_k(y)\ et\ k \leq \min(\sigma(x), \sigma(y))\big) \Rightarrow \big(\sigma(y) < \sigma(x)\big)$

Ce lemme s'interprète de la manière suivante : si deux sommets $x, y$ sont tels que $L_k(x) \prec L_k(y)$ et qu'ils ne sont pas déjà visités à l'étape $k$, alors le sommet $y$ sera visité en premier. Notons en effet que la condition "$x$ n'est pas déjà visité à l'étape $k$" se traduit par $k \leq \sigma(x)$. Pour montrer ce lemme, il suffit de montrer que $\forall j \geq k, L_j(x) \prec L_j(y)$. En effet il en découle immédiatement qu'à l'étape où $x$ ou $y$ est choisi, c'est $y$ qui le sera en premier. On distingue alors 2 cas :

— $L_k(x) \lhd L_k(y)$. Alors la "version forte" de la compatibilité de $\lhd$ avec la concaténation $\cdot$ à droite nous assure que $L_j(x) \prec L_j(y)$ pour tout $k \leq j < n$.

— $L_k(y) = L_k(x) \cdot u$, avec $u \neq \varepsilon$. Dans ce cas, s'il existe $j > k$ tel que $L_j(x) \neq L_k(x)$, on aura *a fortiori* $L_j(x) \lhd L_j(y)$ : si $p = |L_k(x)| + 1$, la $p$-ième lettre de $L_j(y)$ sera bien $>$ à la $p$-ième lettre de $L_j(y)$, car elle a en effet été rajoutée plus tôt.

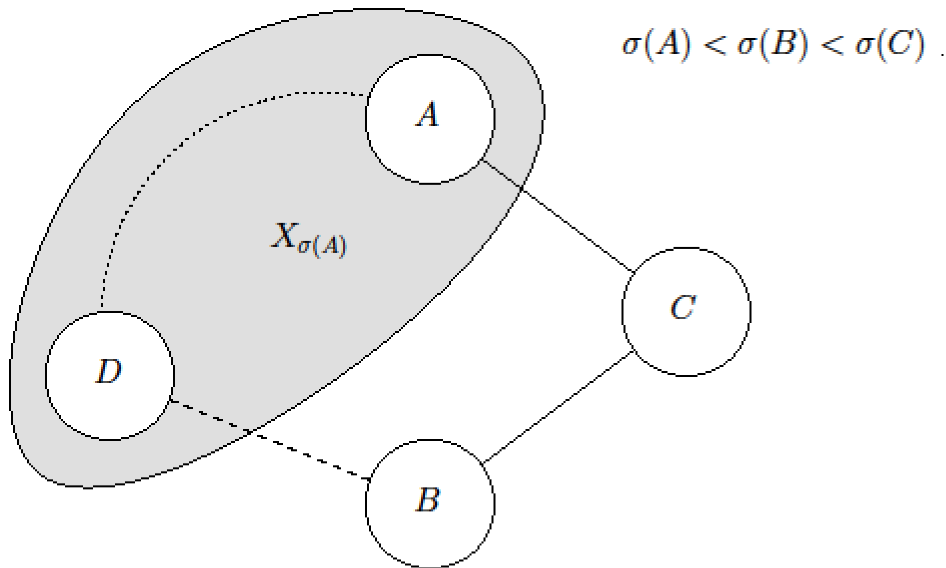



**Fig 3.5**

Retour à la démonstration : on raisonne par l'absurde, et on suppose que $\sigma^{-1}(n-1)$ n'est pas simplicial dans $G$. Alors, si on note par exemple $C = \sigma^{-1}(n-1)$, on peut trouver deux sommets $A$ et $B$ non adjacents qui vérifient $\sigma(A) < \sigma(B) < \sigma(C)$ (cf. **Fig. 3.5.**). On procède alors en 2 étapes :

— 1ère étape : Montrer que $L_{\sigma(A)}(B) = L_{\sigma(A)}(C)$

Soit $k_0 = \min\{k \in [\![0, n-1]\!], L_k(B) \neq L_k(C)\}$.

- 1er cas : $k_0 > \sigma(A)$
- 2ème cas : $k_0 \leq \sigma(A)$ et $L_{k_0}(B) \prec L_{k_0}(C)$

On obtient une contradiction grâce au lemme ($k_0 \leq \min(\sigma(B), \sigma(C))$).

- 3ème cas : $k_0 \leq \sigma(A)$ et $L_{k_0}(C) \prec L_{k_0}(B)$

$L_0(B) = L_0(C)$, on a donc $k_0 > 0$.

Ainsi $L_{k_0-1}(B) = L_{k_0-1}(C) = u$, et $L_{k_0}(B) = u \cdot a_{n-k_0+1}$ et $L_{k_0}(C) = u$.

On note alors $D = \sigma^{-1}(k_0 - 1)$. On a bien $D \in \mathcal{N}(B)$, $D \notin \mathcal{N}(C)$, et $D \in X_{\sigma(A)}$.

On considère alors un chemin sans corde de $D$ à $A$ dans $\left[X_{\sigma(A)}\right]$ $x_1 \dots x_p$, avec $x_1 = D$ et $x_p = A$. Alors $x_1 \dots x_p CBD$ est un trou de longueur $\geq 4$ : contradiction avec $G$ triangulé !

— 2ème étape : On voit alors qu'à l'étape $\sigma(A)$, on obtient $L_{\sigma(A)}(B) \prec L_{\sigma(A)}(C)$ : contradiction avec $\sigma(B) < \sigma(C)$ d'après le lemme ! ■

Étant donné alors qu'un ordre lexicographique $\sigma$ pour $G$ induit aussi un ordre lexicographique pour le graphe $G[V \setminus \{\sigma^{-1}(n-1)\}]$, on en déduit immédiatement la propriété suivante :

**Proposition 3.4.c** : Soit $G = (V, E)$ un graphe triangulé et $\sigma : V \to [\![0, n-1]\!]$ un ordre lexicographique pour $G$. Alors $\sigma^{-1}$ est un schéma d'élimination simplicial.

### 3.5 Affinage de partition, implémentation

Maintenant que nous en avons terminé avec l'aspect théorique du parcours en largeur lexicographique, et mis en évidence le lien tant avec l'algorithme de parcours en largeur qu'avec l'ordre lexicographique sur les mots d'un alphabet totalement ordonné, il reste à aborder la partie pratique de cet algorithme : son implémentation. On pourrait penser que la recherche du sommet d'étiquette maximale à la ligne 6 du programme 5 nécessite un temps de recherche par exemple en $O(n)$, alors qu'il n'en est rien. En effet un des aspects les plus incroyables de cet algorithme, est qu'il peut fonctionner en un temps linéaire en l'ordre et la taille du graphe ($n + m$), via l'utilisation savante d'une structure de donnée particulière (même si la complexité spatiale augmente alors légèrement).

Afin d'obtenir un classement efficace des sommets du graphe, on ne va plus les considérer par leur étiquette, mais par groupes d'une même étiquette. En fait la relation binaire "avoir même étiquette que" est une relation d'équivalence, et l'on va alors considérer la partition de $V$ formée par les classes d'équivalences de cette relation. Il suffira alors de conserver notre partition "triée" par classes de sommets d'étiquette décroissante tout au long de l'algorithme. Le but de l'affinage de partition est de pouvoir mettre à jour la place de $p$ sommets du graphe en un temps $O(p)$. Si l'algorithme Lex_BFS était connu depuis 1976, une implémentation utilisant l'affinage de partition n'a été proposée qu'en 1998.

Formellement, on considère un ensemble fini $X$, et une partition ordonnée de cet ensemble $P = \left\{X_1, \dots, X_p\right\}$. On se donne comme paramètre un sous-ensemble $S \subseteq X$, et on effectue les opérations suivantes : pour toute partie $X_i \in P$, remplacer $X_i$ par $X_i \cap S$ et $X_i \setminus S$ (sauf si $X_i \subseteq S$ ou $X_i \cap S = \emptyset$ ...). La partition $P'$ obtenue est alors une partition plus fine que $P$ (toute partie de $P'$ est incluse dans une partie de $P$). Cette opération peut être effectuée en un temps $O(|S|)$ en considérant $P$ comme une liste doublement chaînée de partie ; et chaque partie est aussi une liste doublement chaînée, dont chaque case pointera également sur l'élément de tête de la partie. On représente cela sur le schéma suivant :

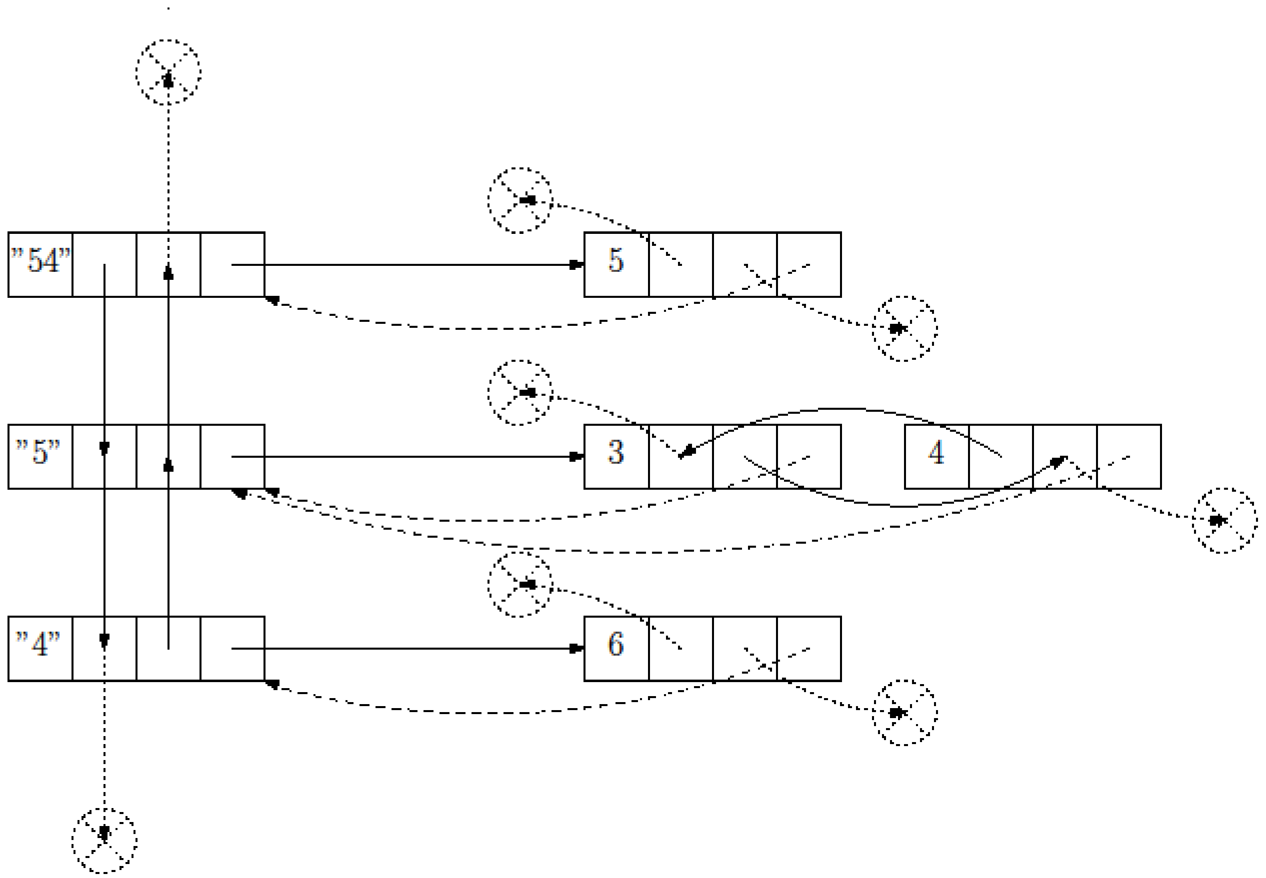



Fig 3.6 – Schéma représentatif d'une partition

La case H (head) comprend un pointeur vers la case H suivante, un pointeur vers la case H précédente, et un pointeur vers le premier élément de la partie correspondante. Chaque élément d'une partie, représenté par une case C (cell), possède un pointeur vers l'élément précédent, un pointeur vers l'élément suivant, et un pointeur vers la tête de la partie (case H).

Ainsi, on considère successivement chaque élément $s \in S$, que l'on peut enlever de sa partie courante et placer dans une nouvelle partie juste avant en un temps constant. La mise à jour se fait bien en un temps $O(|S|)$ ; et dans le cas d'un graphe où l'on affine par rapport à $\mathcal{N}(x)$, où $x$ est le sommet courant visité, il suffit de placer d'abord la partie $X_i \cap \mathcal{N}(x)$ pour conserver les partitions ordonnées par ordre de sommets d'étiquette décroissante.

En Caml, on définira le type partition comme suit :

```
type ('a, 'b) partition = {
    mutable debut: ('a, 'b) cell;
    mutable fin: ('a, 'b) cell}
  and ('a, 'b) cell = Nil | Head of ('a, 'b) header_cell
                          | Case of ('a, 'b) content_cell
  and ('a, 'b) header_cell = {
    mutable content: ('a, 'b) content_cell;
    mutable prev: ('a, 'b) cell;
    mutable next: ('a, 'b) cell;
    mutable flag: 'b}
  and ('a, 'b) content_cell = {e: 'a;
    mutable avant: ('a, 'b) cell;
    mutable apres: ('a, 'b) cell;
    mutable classe: ('a, 'b) header_cell}
;;
```

'a est le type des éléments représentés par les cases.

'b est le type du drapeau ("flag") des cases de tête, utile notamment lors de l'affinage de la partition. On prendra par exemple le type 'b = string pour Lex_BFS, cela permet d'affiner successivement par rapport à chaque sommet de $\mathcal{N}(x)$ en testant si la partie précédente est une partie ajoutée de type $X_i \cap \mathcal{N}(x)$ ou non.

Si l'on se donne enfin 3 fonctions permettant d'agir sur une partition :

```
nouvelle_partition : int -> (int, string) partition = <fun>
retirer_premier_element : ('a, 'b) partition -> unit = <fun>
affinage_partition : ('a, 'b) partition -> 'a list -> unit = <fun>
```

Alors le programme 5 énoncé plus haut peut s'écrire en Caml :

```
let lex_bfs g action =
  let n = vect_length g in
  let partition = nouvelle_partition n in
  for i = 1 to n do
    let a = retirer_premier_element partition in begin
      action a;
      affinage_partition partition g.(a)
    end
  done
;;
```

La fonction a cependant été simplifiée pour des raisons de clarté.

### 3.6 Reconnaissance de graphes triangulés

La production systématique d'un schéma d'élimination simplicial pour un graphe triangulé par l'algorithme Lex_BFS, ainsi que la non-existence d'un tel schéma pour tout graphe qui n'est pas triangulé nous offre un moyen simple de reconnaissance des graphes triangulés. On se donne un graphe quelconque $G$, on effectue un parcours en largeur lexicographique ; si l'ordre obtenu est un schéma d'élimination simplicial, alors $G$ est triangulé, sinon, $G$ n'est pas triangulé.

On peut ainsi produire ce qu'on appelle un algorithme de coloration "robuste" : on lui donne en entrée un graphe quelconque $G$ ; l'algorithme reconnait si $G$ est triangulé, auquel cas il le colore de manière optimale comme indiqué précédemment ; sinon, $G$ n'est pas triangulé et l'algorithme peut appliquer une des heuristiques décrites précédemment (DSATUR par exemple).

Il reste alors à tester si un ordre est simplicial en un temps raisonnable. Pour cela il n'est pas question de tester si les sommets des voisinages des sommets des graphes induits sont tous adjacents, il y a plus rapide. On se donne un graphe quelconque $G$ ainsi qu'une permutation $\sigma : [\![0, n-1]\!] \to V$ des sommets dont on veut tester s'il s'agit d'un schéma d'élimination simplicial.

Notons $PN(x) = \{y \in \mathcal{N}(x), \sigma^{-1}(y) < \sigma^{-1}(x)\}$, l'ensemble des "voisins précédents" (Previous Neighbors), et $p(x) \in PN(x)$ tel que $\sigma^{-1}\big(p(x)\big) = \max(\sigma^{-1}\langle PN(x)\rangle)$ s'il existe, le sommet de $PN(x)$ "le plus à droite". Dire que $\sigma$ est un ordre d'élimination simplicial signifie exactement que pour tout $x$, $PN(x)$ est une clique. On utilise alors la propriété suivante :

**Proposition 3.6** : Soit $i \in [\![0, n-1]\!]$ et $x = \sigma(i)$. On suppose que $\forall j \in [\![0, i-1]\!], PN\big(\sigma(j)\big)$ est une clique. Alors $(PN(x)\ est\ une\ clique) \Leftrightarrow \Big(PN(x) \setminus p(x) \subseteq PN\big(p(x)\big)\Big)$

**Démonstration** : Notons déjà qu'on peut considérer l'équivalence comme vraie si $p(x)$ n'est pas défini : à ce moment là on aura $PN(x) = \emptyset$, et donc l'assertion $PN(x) \setminus p(x) \subseteq PN(p(x))$ sera vraie quelle que soit la valeur de $p(x)$ prise par l'algorithme. On notera ensuite $a - b$ pour dire que deux sommets $a$ et $b$ sont adjacents.

$\Rightarrow$ : On suppose que $PN(x)$ est une clique.

On a alors $\forall y \in PN(x) \setminus p(x), y - p(x)$. Or on a aussi, $\forall y \in PN(x) \setminus p(x), \sigma^{-1}(y) < \sigma^{-1}(p(x))$ par définition de $p(x)$ ; donc on trouve bien $PN(x) \setminus p(x) \subseteq PN(p(x))$.

$\Leftarrow$ : On suppose $PN(x) \setminus p(x) \subseteq PN(p(x))$.

On a $\sigma^{-1}(p(x)) < i$, donc $PN(p(x))$ est une clique. D'où $PN(x) \setminus p(x)$ est une clique.

Reste à montrer que $\forall y \in PN(x) \setminus p(x), y - p(x)$. Découle du fait que $y \in PN(p(x))$ ■

Pour représenter $PN(x)$, on utilise une matrice $n \times n$, qui indique $PN(x)(y) = true$ si et seulement si $y \in PN(x)$. On obtient alors l'algorithme de reconnaissance suivant :

**Programme 6** Reconnaissances des graphes triangulés

**Entrée** – Un graphe quelconque $G = (V, E)$

**Sortie** – Un booléen : *true* si $G$ est triangulé, *false* sinon

```
1	n := |V|;
2	# Numero : V → ⟦0, n − 1⟧ #
3	Numero := Lex_BFS (Graphe : G) (Source : 0);
4	# Numero_inverse : ⟦0, n − 1⟧ → V #
5	Numero_inverse := Tableau (Taille : n) (Defaut : 0);
6	PN := Matrice (Hauteur : n) (Largeur : n) (Defaut : false);
7	Pour x de 0 à n-1 faire
8		Numero_inverse.(Numero.(x)) = x
9	Fin Pour;
10	Pour i de 0 à n-1 faire
11		x :=Numero_inverse.(i);
12		px := Numero_inverse.(0);
13		# Première étape : Construction de PN(x), et recherche de p(x)#
14		Pour y dans N(x) faire
15			Si Numero.(y) < i faire
16				PN.(x).(y) ← true;
17				Si Numero.(y) > Numero.(px) faire
18					px := y
19				Fin Si
20			Fin Si
21		Fin Pour
22		# Deuxième étape : Vérification de PN(x) \ p(x) ⊆ RN(p(x)) #
23		Pour y dans N(x) faire
24			Si PN.(x).(y) & y ≠ px & PN.(px).(y) = false faire
25				Renvoyer false
26			Fin Si
27		Fin Pour
28	Fin Pour;
29	Renvoyer true
```

Là encore, on voit que le temps d'exécution est clairement $O(n + m)$. En revanche la complexité spatiale, du fait de l'utilisation d'une matrix $n \times n$, est ici $O(n^2)$.

# 4 Résultats expérimentaux

## 4.1 Complexité temporelle des algorithmes

Il est temps de passer aux résultats expérimentaux. On a implémenté les différents algorithmes présentés en Caml. On réalisé également différentes implémentations des algorithmes proposés. Les fonctions en Caml sont les suivantes :

```
color_welsh_powell : int list vect -> int = <fun>
color_welsh_powell2 : int list vect -> int = <fun>
dsatur : int list vect -> int = <fun>
dsatur2 : int list vect -> int = <fun>
dsatur3 : int list vect -> int = <fun>
color_lex : int list vect -> int = <fun>
```

On a réalisé deux implémentations de l'algorithme de Welsh & Powell, la première usant du tri par fusion sur les listes implémenté en Caml dans la librairie `sort`, la deuxième utilisant un tri par dénombrement (censé être de complexité linéaire). La première version de Dsatur utilise la remarque faite en 2.2 sur l'utilisation d'une matrice ; la deuxième version effectue les mises à jour sans utiliser une telle astuce ; et la troisième utilise les structures de données d'affinage de partition utilisées pour Lex_BFS.

Les essais on été menés sur des graphes pseudo-aléatoire de densité moyenne ½. L'allure des courbes étant sensiblement la même pour des graphes pseudo-aléatoire de densité différente, on n'a reporté ici que les résultats obtenus sur les graphes de densité moyenne ½. Tous les tests présentés par la suite ont été menés sur un ordinateur fonctionnant sous **Windows XP SP3**, doté d'un **Intel® Core™ 2 Duo E6400 2.13 GHz** et de **2.00 Go** de RAM.

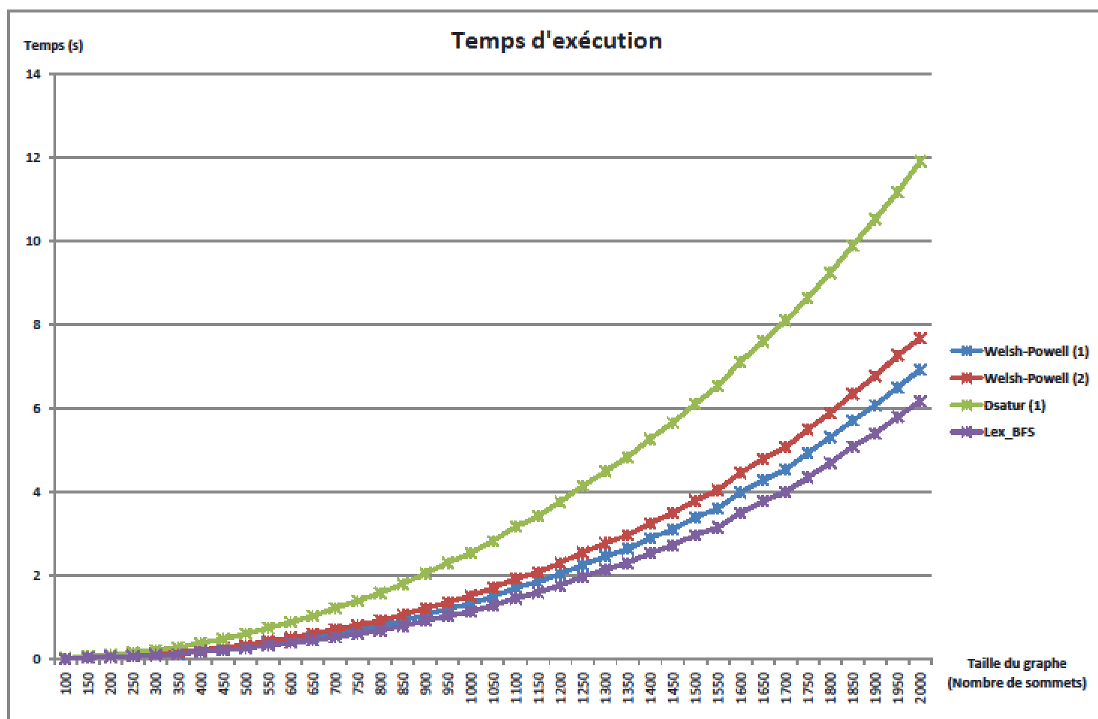



Fig 4.1.1 – Résultats sur des graphes pseudos-aléatoires

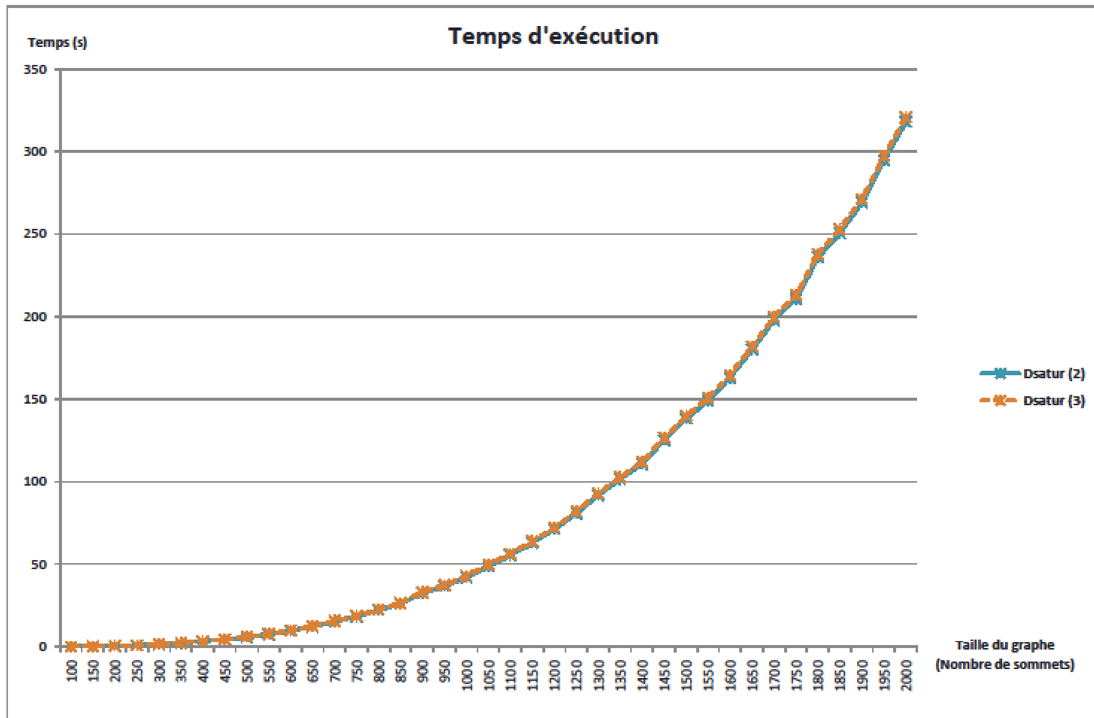



Fig 4.1.2 – Résultats sur des graphes pseudos-aléatoires (suite)

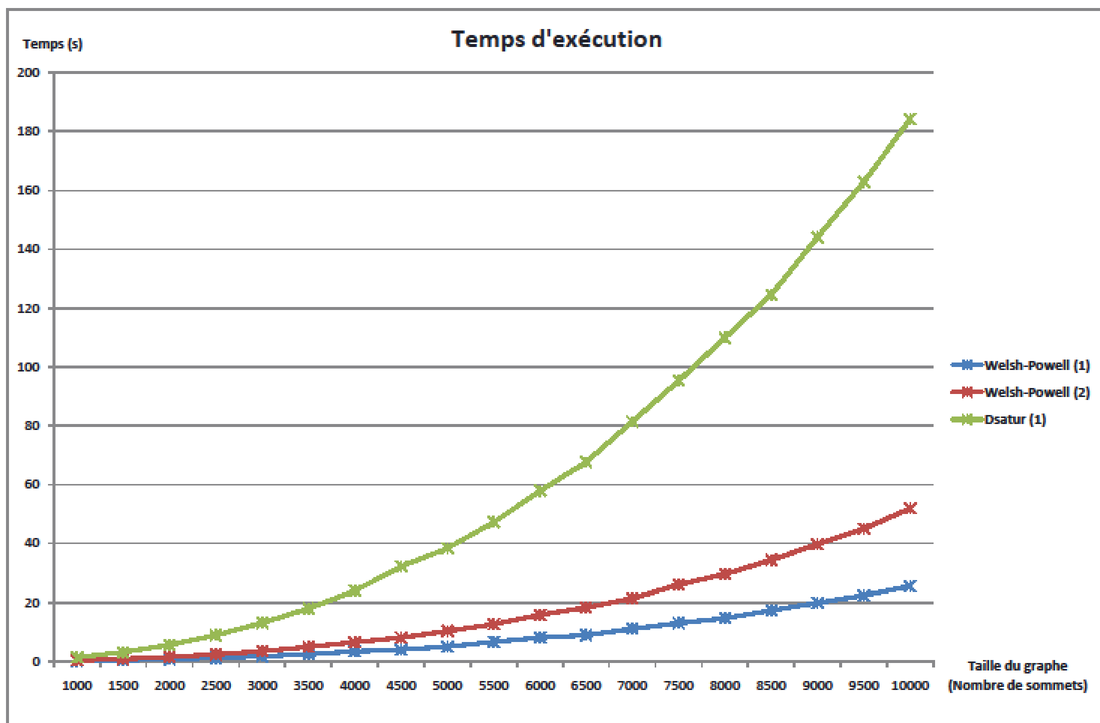


**Fig 4.1.3** – Résultats sur des graphes pseudos-aléatoires (fin)

Des problèmes de dépassement de capacité de mémoire n'ont pas permis de mener à bien les tests sur les fonctions `dsatur2`, `dsatur3` et `color_lex` pour une série de graphes au-delà des 1000 sommets.

Nous avons tenté de réutiliser les techniques mises en place pour l'implémentation de Lex_BFS pour l'algorithme `dsatur3`. Malheureusement on a pu constaté que cet effort s'est avéré vain : la complexité temporelle de l'algorithme est la même que pour `dsatur2` ! (C'est-à-dire pas terrible). Cette augmentation s'explique sans doute par une trop grande complexité spatiale, ainsi que des mises à jours qui seraient plus nombreuses que ce à quoi on pourrait s'attendre au départ.

Enfin, on notera que le tri par fusion utilisé pour `color_welsh_powell`, de complexité semi-linéaire s'avère en pratique deux fois plus efficace que l'algorithme de tri par dénombrement pour liste que nous avons pu implémenter. Là encore le double parcours de liste effectué par l'algorithme de tri par dénombrement (pour chercher les bornes du tableau utiliser pour ranger les éléments) et l'utilisation également d'un tel tableau explique sans doute la baisse de performances.

### 4.2 Efficacité moyenne

Bien que de tels essais ne soient pas nécessairement très significatifs sur des graphes pseudo-aléatoires, on a recueilli également les informations sur les colorations obtenues lors des tests de complexité temporelle. On peut donc voir la différence d'efficacité en terme de nombre de couleurs utilisées par les différents algorithmes sur les deux séries de graphes pseudo-aléatoires. Cela confirme notamment que DSATUR semble plus efficace en général que Welsh & Powell, et qu'une coloration séquentielle selon un ordre lexicographique (`color_lex`) fourni également un résultat plus mauvais que Welsh & Powell.

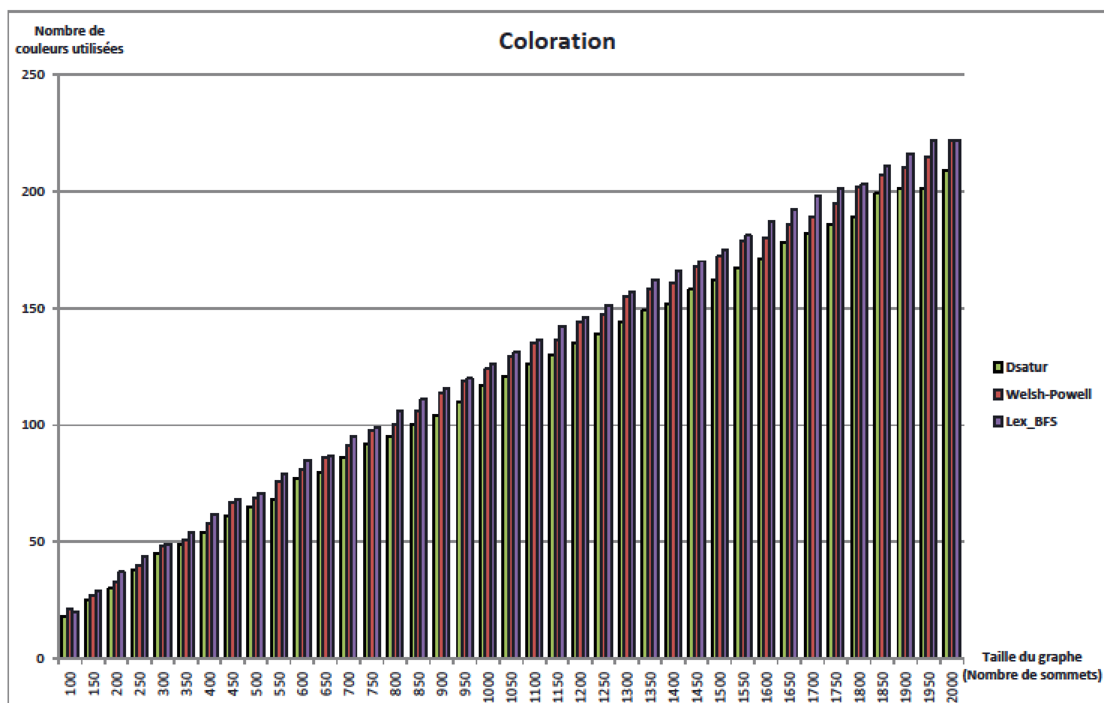



**Fig 4.2.1** – Colorations obtenues sur des graphes pseudos-aléatoires

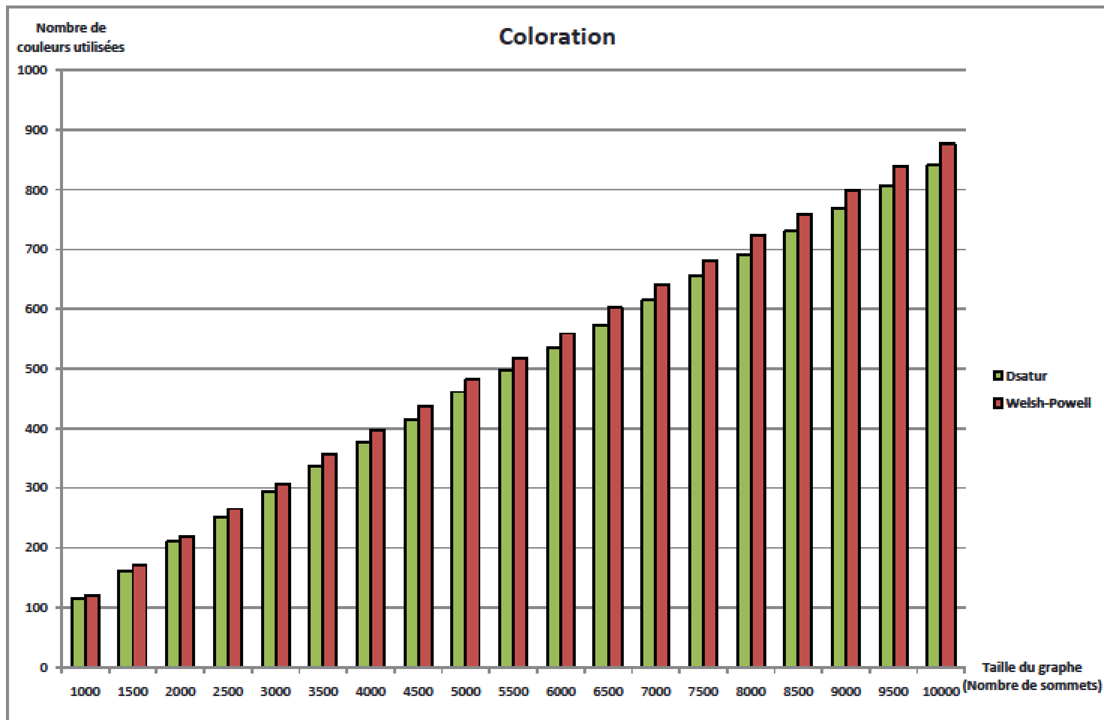



**Fig 4.2.2** – Colorations obtenues sur des graphes pseudos-aléatoires

### 4.3 Le benchmark de DIMACS

DIMACS, pour Center for Discrete Mathematics and Theoretical Computer Science, est issu d'une collaboration entre les universités de Rutgers et de Princeton aux États-Unis. Fondé en 1989, il est dédié au développement théorique et aux applications pratiques liés aux mathématiques discrètes et à l'informatique théorique.

DIMACS parraine des défis d'implémentations informatique ayant pour but la mise en place d'algorithmes pratiques destinés à faire face à des problèmes concrets. Il y a eu pour l'instant 9 défis à relever, et l'un des aspects du défi de l'année 1992 était consacré à la coloration de graphe. De fait on dispose à ce jour de graphes de benchmark, de référence, utilisés lors de ce défi et qui correspondent chacun à divers problèmes concrets rencontrés en informatique et en modélisation.

Les problèmes modélisés par ces graphes de benchmark sont multiples : allocation de registres, planification d'horaires, réseaux optiques, etc. Et l'essai sur cette banque de donnée d'algorithme de coloration est plus significative, en terme d'optimalité de la coloration obtenue, que des essais menés sur des graphes pseudo-aléatoires. En effet certains de ces graphes sont fournis avec leur nombre de coloration, ce qui permet de mieux mesurer l'efficacité de nos algorithmes.

Les graphes sont disponibles sous forme de fichier texte (.col). De là il est facile de construire le graphe en Caml en lisant le fichier ligne par ligne. On a reporté dans le tableau suivants les mesures effectuées sur les algorithmes décrits plus hauts :

**Tableau 1 - Comparaison de différents algorithmes sur des graphes de la librairie DIMACS**

| Graphe | Nombre de : | | | Nb Couleurs | | Temps d'exécution (s) | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | Sommets | Arêtes | Coloration | W.P. | Dsatur | Welsh-Powell | | Dsatur | | |
| | n | m | χ(G) | | | 1 | 2 | 1 | 2 | 3 |
| inithx.i.1.col | 864 | 18707 | 54 | 54 | 54 | 0.047 | 0.031 | 0.359 | 1.297 | 0.125 |
| mulsol.i.4.col | 185 | 3946 | 31 | 31 | 31 | 0 | 0 | 0.031 | 0.125 | 0.016 |
| zeroin.i.1.col | 211 | 4100 | 49 | 49 | 49 | 0.016 | 0 | 0.031 | 0.187 | 0.016 |
| school1_nsh.col | 352 | 14612 | ??? | 34 | 27 | 0 | 0.015 | 0.094 | 0.547 | 0.047 |
| jean.col | 80 | 508 | 10 | 10 | 10 | 0 | 0 | 0 | 0.015 | 0 |
| queen11_11.col | 121 | 3960 | 11 | 17 | 15 | 0 | 0 | 0.015 | 0.11 | 0.015 |
| myciel4.col | 23 | 71 | 5 | 5 | 5 | 0 | 0 | 0 | 0 | 0.016 |
| myciel7.col | 191 | 2360 | 8 | 8 | 8 | 0 | 0 | 0.016 | 0.047 | 0.015 |
| mug100_1.col | 100 | 166 | 4 | 4 | 4 | 0 | 0 | 0 | 0.015 | 0 |
| wap05a.col | 905 | 43081 | ??? | 51 | 51 | 0.032 | 0.046 | 0.454 | 2.515 | 0.156 |
| wap07a.col | 1809 | 103368 | ??? | 52 | 46 | 0.093 | 0.094 | 1.547 | 6.562 | 0.391 |
| qg.order40.col | 1600 | 62400 | 40 | 64 | 40 | 0.062 | 0.047 | 1.094 | 3.531 | 0.25 |
| qg.order100.col | 10000 | 990000 | 100 | 128 | 100 | 0.797 | 0.766 | 39.375 | 142.219 | 4.062 |

Description des différents problèmes représentés :

- Allocation de registre : `inithx.i.1.col`, `mulsol.i.4.col` et `zeroin.i.1.col`.

Lors de la production de code, à l'étape de la compilation, il faut allouer aux différentes variables du programme des registres du processeur. Ces registres, emplacements de mémoires internes au processeur, permettent d'effectuer des opérations plus rapides que sur des objets stockés dans la mémoire vive. Ils sont malheureusement en nombre très limité, donc il est essentiel de les allouer avec parcimonie.

On représente donc le problème par un graphe dont les sommets sont les variables du programme. Deux sommets sont reliés entre eux si les variables correspondantes interfèrent, c'est-à-dire sont susceptibles d'être utilisées en même temps au cours de l'exécution. Deux variables qui n'interfèrent pas se voir allouer le même registre, puisqu'il n'y aura pas de risque de conflit lors de l'exécution. Colorer le graphe avec $k$ couleurs revient alors à allouer $k$ registres différents aux variables du programme.

- Planification d'horaires : `school1_nsh.col`.

On imagine qu'il faut créer les emplois du temps de plusieurs classes et de professeurs, de sorte que chaque élève puisse suivre les cours pour lesquels il est inscrit, et chaque professeur puisse dispenser ses cours à ses élèves. Il faut aussi minimiser le nombre de créneaux horaires utilisés car ceux-ci sont limités (un établissement scolaire qui ferme après 18h par exemple).

On représente cela par un graphe où les sommets sont des cours (un couple $(classe, prof)$), et deux sommets sont adjacents si les cours qu'ils représentent sont dispensés par le même professeur ou à la même classe. Colorer le graphe avec un minimum de couleur revient donc à allouer à ces différents cours des créneaux horaires compatibles avec les aspirations de chacun.

- Graphes de Livres : `jean.col`.

Créé par Donald Knuth, ces graphes représentent les différentes relations des personnages d'un livre. Les sommets du graphes sont les protagonistes de l'œuvre, et deux sommets sont adjacents si les personnages correspondants se rencontrent au cours de l'histoire. Ici l'œuvre qui est représentée est *Les Misérables* de Victor Hugo.

- Graphes de Reines : `queen11_11.col`.

On peut considérer cela comme une généralisation du problème des huits dames ; ces graphes modélise le problème suivant : étant donné un échiquier de taille $n \times n$, est-il possible d'y placer $n$ ensembles de $n$ reines de sorte que deux reines d'ensembles différents ne puissent pas se prendre entre elles ? (En respectant les règles classiques du jeu d'échec, deux reines d'ensembles différents ne sont donc jamais sur la même ligne, colonne, ou diagonale).

Le graphe considéré est alors un graphe d'ordre $n^2$, dont les sommets sont les cases de l'échiquier. Deux sommets sont adjacents si les cases correspondantes sont sur une même ligne, une même colonne, ou une même diagonale. La réponse à la question posée est ainsi oui si et seulement si le graphe obtenu a un nombre chromatique égal à $n$. Notons qu'un graphe de reines n'est jamais $(n-1)$-colorable.

- Graphes de Mycielski : `myciel4.col` et `myciel7.col`.

Ces graphes sont sans triangle (nombre de clique égal à 2).

- `mug100_1.col` : Un graphe presque 3-coloriable, mais qui possède une clique de cardinal 4.
- Réseaux Optique : `wap05a.col` et `wap07a.col`.

Deux graphes correspondant à un problème concret dans la conception de réseaux optiques : chaque sommet correspond à un chemin lumineux dans le réseau, et les arêtes signifient que deux chemins s'intersectent. Allouer des couleurs différentes aux sommets du graphe, de sorte que deux sommets adjacents n'aient pas la même couleur revient par exemple à utiliser différentes ondes lumineuses (de longueurs d'ondes différentes par exemple), pour véhiculer l'information dans les chemins que ces sommets représentent (et deux faisceaux différents pourront se croiser sans risque de brouillage d'information).

- Carrés Latins : `qg.order40.col` et `qg.order100.col`.

Un carré latin est une matrice $n \times n$, remplie de $n$ éléments distincts, et dont chaque élément apparait une et une seule fois dans chaque ligne et chaque colonne de la matrice. Typiquement les valeurs prises par la matrice sont les entiers $\{1, \dots, n\}$. On notera le lien avec d'autres objets comme les carrés magiques ou le sudoku. Il existe des carrés latins de taille $n$ pour tout $n \in \mathbb{N}$. En effet il suffit de considérer la matrice :

$$\begin{bmatrix} 1 & n & \cdots & 2 \\ 2 & 1 & \cdots & \vdots \\ \vdots & \vdots & \ddots & n \\ n & n-1 & \cdots & 1 \end{bmatrix}$$

On modélise donc la matrice par un graphe dont les sommets sont les cases de la matrice, et deux sommets sont adjacents si les cases sont dans la même ligne ou dans la même colonne. Colorer ce graphe avec $c$ couleurs revient donc à placer $c$ éléments distincts dans la matrice de sorte qu'il n'y ait jamais deux fois le même élément sur une même ligne ou une même colonne. Trouver toutes les $n$-colorations d'un tel graphe d'ordre $n^2$ revient donc à déterminer tous les carrés latins de taille $n$.

Reste enfin à interpréter les résultats obtenus :

On peut voir que dans la plupart des cas, WP (pour Welsh & Powell) et DSATUR fournissent la coloration optimale du graphe considérer. Cependant lorsqu'il y a des différences, DSATUR est toujours plus performant que WP (`school1_nsh.col`, `queen11_11.col` et `wap07a.col` par exemple). Lorsque le graphe présente une structure plus "régulière" (des symétries par exemple avec `qg.order40.col`), DSATUR fournit encore une meilleures coloration que WP.

Au niveau des temps d'exécution, les graphes présentés étant de faible taille (généralement moins de 100 000 arêtes, et presque un million pour `qg.order100.col`), les temps obtenus sont assez faibles et ne présentent pas de grandes différences. Toutefois on peut observer que pour le dernier exemple (`qg.order100.col`), le temps d'exécution légèrement supérieur de DSATUR devient nettement plus visible. Encore plus visible sur ce même graphe est la grande rapidité de `dsatur3`, alors qu'on avait observé des temps d'exécutions exécrables (du même ordre de grandeur que `dsatur2`) sur des graphes pseudo-aléatoires. On peut donc penser que la forte symétrie des graphes de type `qg.order40.col` est propice à l'utilisation de la structure de donnée utilisée pour implémenter `dsatur3` : l'affinage de partition.

# Conclusion et perspectives

En conclusion de ce tour d'horizon, on insistera sur les différences d'efficacités obtenues, et sur la nécessité d'utiliser des heuristiques efficaces. On a bien mis en valeur la complexité du problème de coloration et le besoin de recourir à des heuristiques rapides dans le cas général. On notera le fort développement des activités de recherches consacrées à la résolution pratique, et optimale, en un temps polynomial, du problème de coloration pour certaines sous-classes de graphes parfaits (les graphes triangulés donc, mais pas seulement).

On ouvrira finalement en appuyant sur le caractère fondamental de ce genre de recherche : la recherche de solutions optimales pour des cas particuliers de graphe de plus en plus nombreux permet non seulement le développement de tout un panel d'algorithme auxquels on finit toujours par trouver d'autres applications ; mais cela permet en plus de se rapprocher d'un autre objectif : celui d'apporter une solution au problème P=NP, ou encore de servir directement dans les multiples applications que comporte la coloration de graphe.

On notera également que le problème de coloration simple abordé n'est pas exhaustif, et qu'il peut s'étendre de multiples façons : coloration avec contraintes (nombre limité de couleurs, sommets avec couleurs imposées, etc.), multicoloration (allocation d'une liste de couleurs à chaque sommet), etc.

# Bibliographie


- *Coloration de graphes*, cours de l'université de Nice Sophia Antipolis (Frederic Havet)

- *Cours d'algorithmique des graphes du MPRI*, de Michel Habib

- *Algorithmic Aspects of Vertex Elimination on Graphs*, par Donald J. Rose, R. Endre Tarjan et George S. Lueker

- *LexBFS and Partition Refinement*, par Michel Habib, Ross McConnell, Christophe Paul et Laurent Viennot


# Webographie


- DSATUR, page web par Philippe Rolland :
http://prolland.free.fr/Cours/Cycle2/Maitrise/GraphsTheory/TP/PrgGraphDsat/dsat.html

- DIMACS, les graphes de benchmark :
http://mat.gsia.cmu.edu/COLOR02/