Allocation de Fréquences par Coloration de Graphes

Table des matières

1 Généralités **2**
1.1 Le problème d'allocation de fréquences 2
1.2 Graphes, définitions 3
1.3 Le problème de coloration 3

2 Graphes quelconques : des algorithmes d'approximation **4**
2.1 Coloration séquentielle : l'algorithme glouton 4
2.2 Deux heuristiques différentes 5
2.2.1 Welsh & Powell 5
2.2.2 DSATUR 5

3 Graphes triangulés : un algorithme linéaire de coloration optimale **6**
3.1 Généralités sur les graphes triangulés 6
3.2 Parcours en largeur lexicographique - LexBFS 6
3.3 Affinage de partition 7

4 Résultats expérimentaux **8**
4.1 Sur des graphes pseudo-aléatoires 8
4.2 Sur des graphes de DIMACS 10

Conclusion et perspectives **12**

Références **12**

1 Généralités

1.1 Le problème d'allocation de fréquences

Imaginons que l'on veuille construire un réseau d'antennes radios. Il faudra réserver certaines fréquences de communication, que l'on attribuera à nos antennes afin qu'elles puissent communiquer entre elles. Il faut aussi minimiser le nombre de fréquences à réserver, car chaque réservation a un coût. Une première modélisation du problème, en radio AM par exemple, montre que si deux antennes "assez proches" géographiquement émettent dans des fréquences trop voisines l'une de l'autre, leur spectres en fréquences se recouvrent et il devient impossible d'extraire l'information nécessaire du signal reçu. On dit alors que les antennes interfèrent.

On modélise ainsi le réseau d'antenne par un graphe, appelé *graphe d'interférences*, dont les sommets sont les antennes, et les arêtes relient deux antennes qui interfèrent entre elles. Les fréquences à allouer correspondent à des couleurs (représentées par des entiers) avec lesquelles nous colorons les sommets du graphe. La coloration doit être une coloration propre : deux sommets adjacents ne peuvent recevoir la même couleur. Le graphe sera de plus non-orienté, la relation "interfère" étant supposé symétrique.

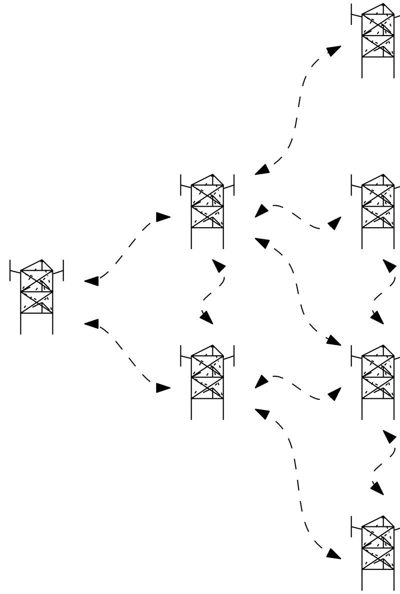


FIGURE 1 – Un réseau d'antennes

1.2 Graphes, définitions

Supposons maintenant le graphe d'interférence du réseau connu. Rappelons quelques définitions et notations utilisées par la suite :

- Un *graphe* est un couple $G = (V, E)$ où V est un ensemble fini, ses éléments sont les *sommets* du graphe, et $E \subseteq V^2$, ses éléments sont les *arcs* du graphe.
  Ici les graphes sont *simples* ($\forall a \in V, (a, a) \notin E$) et *non-orientés* ($\forall a, b \in V,\ (a, b) \in E \iff (b, a) \in E$. Les arcs sont alors appelés des *arêtes*.
  L'*ordre* (resp. la *taille*) d'un graphe est le nombre de ses sommets (resp. arêtes).
  Par la suite nous prendrons $V = [\![0, n-1]\!]$, où $n = |V|$, et on notera $m = |E|$.
- Deux sommets a et b sont dit *adjacents* ou *voisins* 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 voisins à a, ou *voisinage* de a. On note $d(a) = |\mathcal{N}(a)|$ le nombre de voisins de a, ou *degré* de a.
- $H = (V', E')$ est un *sous-graphe* de G si $V' \subseteq V$ et $E' \subseteq E$. Soit $S \subseteq V$, on note $G[S]$ le graphe $(S, E \cap (S \times S))$, appelé *sous-graphe* de G *induit*, ou *engendré*, par S.
- Une *chaîne* de *longueur* k est une suite finie de sommets deux à deux adjacents $v_0 v_1 \dots v_k$, i.e. telle que $\forall i \in [\![1, k]\!], (v_{i-1}, v_i) \in E$. 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.

1.3 Le problème de coloration

Soit $G = (V, E)$ un graphe simple non-orienté. Une application $\sigma : V \to \mathbb{N}$ est appelée 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. Soit $k = |\sigma \langle V \rangle|$, on dit alors que σ est une *k-coloration propre* de G, et que G est *k-colorable*. Le plus petit entier k pour lequel G est k-colorable est appelé *nombre chromatique* de G, noté $\chi(G)$. Nous recherchons ainsi une coloration optimale de G, c'est-à-dire une $\chi(G)$-coloration propre de G.

Allouer des fréquences au réseau d'antennes revient alors à colorer proprement son graphe d'interférences. Notons qu'en raison de contraintes géographiques, technologiques, ou autres (présence de montagnes, etc.), nous ne pouvons pas faire d'hypothèses *a priori* sur la structure du graphe d'interférences.

Nous donnerons donc un exemple de méthode employée pour colorer un graphe d'une manière plus générale. Trouver une coloration optimale d'un graphe quelconque étant un problème NP-Complet, nous exposerons tout d'abord des algorithmes de coloration non-optimale dans le cas général, ainsi qu'un algorithme optimal dans un cas particulier.

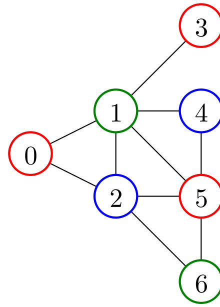


Figure 2 – Un graphe d'interférence coloré

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. Nous obtenons ainsi une coloration propre du graphe, pas forcément optimale, et qui dépend fortement de l'ordre de visite des sommets.

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) (Défaut : −1) ;
3:  Pour i = 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 : ? ? ?) (Défaut : vrai) ;
7:     Pour y ∈ 𝒩 (x) faire
8:        Si Couleur.(y) ≠ -1 alors
9:           Libre.(Couleur.(y)) ← faux
10:       Fin Si
11:    Fin Pour
12:    index := 0 ;
13:    Tant Que Libre.(index) = faux 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 complexité temporelle de l'algorithme est $O\,(n + m)$.

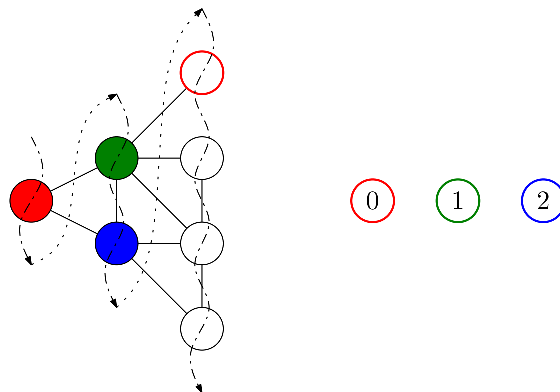


Figure 3 – Coloration séquentielle d'un graphe

2.2 Deux heuristiques différentes

2.2.1 Welsh & Powell

Il s'agit maintenant d'utiliser l'algorithme de coloration séquentiel avec un ordre judicieux, en vue d'obtenir une coloration propre la plus "acceptable" possible. L'algorithme de Welsh & Powell consiste ainsi à colorer séquentiellement le graphe en visitant les sommets par ordre de degré décroissant. L'idée est que les sommets ayant beaucoup de voisins seront plus difficiles à colorer, et donc il faut les colorer en premier. La complexité de l'algorithme devient $O(n \ln n + m)$ en utilisant un tri par comparaison, mais reste $O(n + m)$ avec un tri par dénombrement.

Cependant on peut parfois aboutir aux pires colorations possibles ($\frac{n}{2}$ au lieu de 2). L'heuristique DSATUR propose une amélioration du principe de l'algorithme de Welsh & Powell afin d'éviter ce problème.

2.2.2 DSATUR

Ici, l'idée est que les sommets difficiles à colorer sont ceux qui ont le plus de voisins de couleurs différentes, puis ceux qui ont le plus de voisins tout court. Concrètement, on définit le *degré de saturation* d'un sommet a, noté $DSAT\left(a\right)$ comme suit :

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

Ou encore, si σ désigne la fonction de coloration partielle, $DSAT\left(a\right) = \begin{cases} d\left(a\right) \ \ si \ \sigma\left\langle \mathcal{N}\left(a\right)\right\rangle = \emptyset \\ \left|\sigma\left\langle \mathcal{N}\left(a\right)\right\rangle\right| \ \ sinon \end{cases}$

Après initialisation de $DSAT\left(a\right)$ à $d\left(a\right)$ pour tout a, on répète alors les opérations suivantes :

1. Prendre un sommet non coloré x de $DSAT$ maximum.
2. Colorer x avec la plus petite couleur disponible.
3. Mettre à jour $DSAT\left(y\right)$ pour $y \in \mathcal{N}\left(x\right)$.

La complexité temporelle de l'algorithme dépend de l'implémentation effectuée. Elle peut être $O\left(n^2\right)$ ou $O\left(n^2 + nm\right)$ selon la structure utilisée.

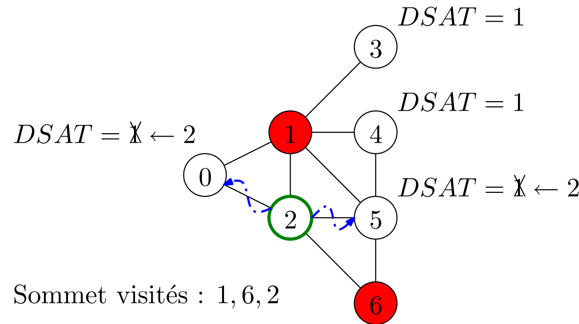


Figure 4 – Une étape de l'algorithme DSATUR

3 Graphes triangulés : un algorithme linéaire de coloration optimale

3.1 Généralités sur les graphes triangulés

Nous allons maintenant donner un algorithme optimal qui permet de colorer une classe de graphe bien particulière : les graphes triangulés. Nous présentons ici les définitions et propriétés utiles à cet algorithme.

- Un graphe $G = (V, E)$ est dit *triangulé* s'il ne contient pas de trou de longueur ≥ 4.
- Un sommet a de G est *simplicial* si tous ses voisins sont adjacents 2 à 2.
- Un *schéma* (ou *ordre*) *d'élimination simplicial* est une permutation $\sigma\ : [\![0, n-1]\!] \to V$ telle que pour tout i dans $[\![0, n-1]\!]$, $\sigma(i)$ soit simplicial dans $G[\sigma\langle[\![0, i]\!]\rangle]$.

On peut alors démontrer la caractérisation suivante, sur laquelle repose l'algorithme LexBFS :

Propriété 1. *G est triangulé si et seulement si il admet un schéma d'élimination simplicial.*

L'utilité d'une telle caractérisation est manifeste, car on peut montrer qu'une coloration séquentielle selon un tel ordre offre une coloration optimale. Le problème consiste donc à trouver efficacement un tel schéma dans le cas des graphes triangulés, c'est le but de l'algorithme LexBFS.

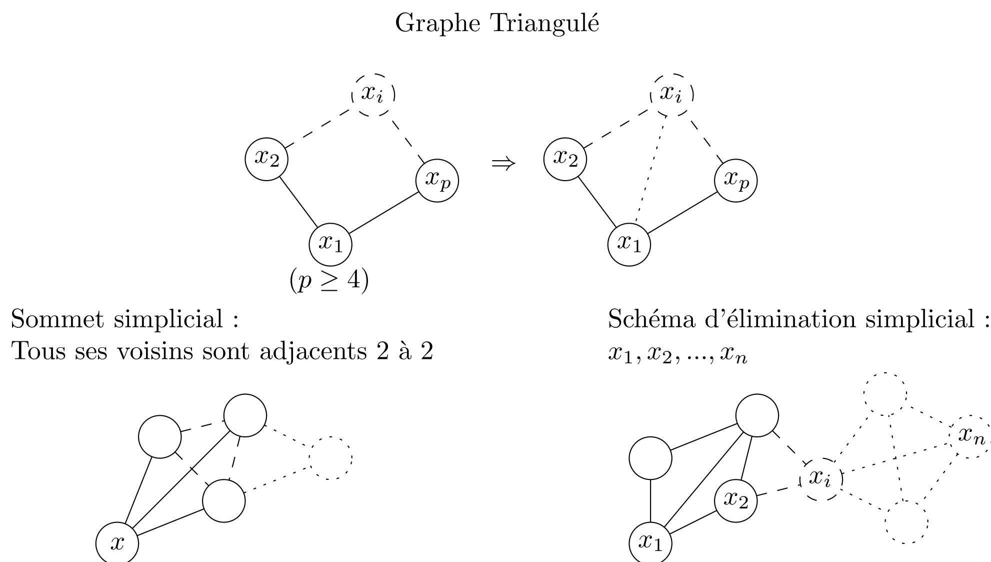


FIGURE 5 – Illustrations

3.2 Parcours en largeur lexicographique - LexBFS

Il se présente comme une variante du parcours en largeur classique de graphe (dit BFS pour "Breadth First Search"). Considérons l'alphabet $A = [\![1, n]\!]$, et munissons l'ensemble des mots sur A de l'ordre lexicographique $\preccurlyeq$, défini intuitivement comme étant l'ordre du dictionnaire. On associe alors à chaque sommet x une étiquette $L(x) \in A^*$, initialement égale à ε, où ε désigne le mot vide de A^*.

On effectue ensuite les opérations suivantes :

Pour i **variant de** 0 **à** $n-1$:

1. Prendre un sommet x d'étiquette maximale pour $\preccurlyeq$ parmi les sommets non visités.
2. Affecter le numéro i à x.
3. Effectuer la mise à jour $L(y) \leftarrow L(y) \cdot (n-i)$ pour $y \in \mathcal{N}(x)$

Programme 2 Parcours en Largeur Lexicographique

Entrée : Un graphe quelconque $G = (V, E)$, un sommet source $s \in V$
Sortie : Une bijection $\sigma : V \to [\![0, n-1]\!]$

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

Nous obtenons alors un ordre $\sigma : V \to [\![0, n-1]\!]$ sur les sommets du graphe, appelé "ordre lexicographique", qui vérifie la propriété suivante :

Propriété 2. *Si G est triangulé, alors $\sigma^{-1}(n-1)$ est simplicial.*

Il en découle, en enlevant successivement le dernier sommet visité, que σ^{-1} est un schéma d'élimination simplicial si G est triangulé.

3.3 Affinage de partition

Outre la possibilité de colorer de manière optimale un graphe triangulé, LexBFS offre aussi un moyen de reconnaissance de graphe triangulé : il suffit de tester si l'ordre obtenu est un schéma d'élimination simplicial. Plus intéressant encore : il peut s'implémenter de manière à avoir une complexité linéaire.

Afin de pouvoir extraire un sommet d'étiquette maximale en temps constant, nous maintenons en place une partition ordonnée des sommets non visités selon leurs étiquettes. La structure utilisée est une liste doublement chaînée de listes doublement chaînées, chaque cellule étant dotée d'un pointeur vers la "cellule de tête" contenant sa partie.

Le type de donnée utilisé en Caml Light est le suivant :

```
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}
;;
```

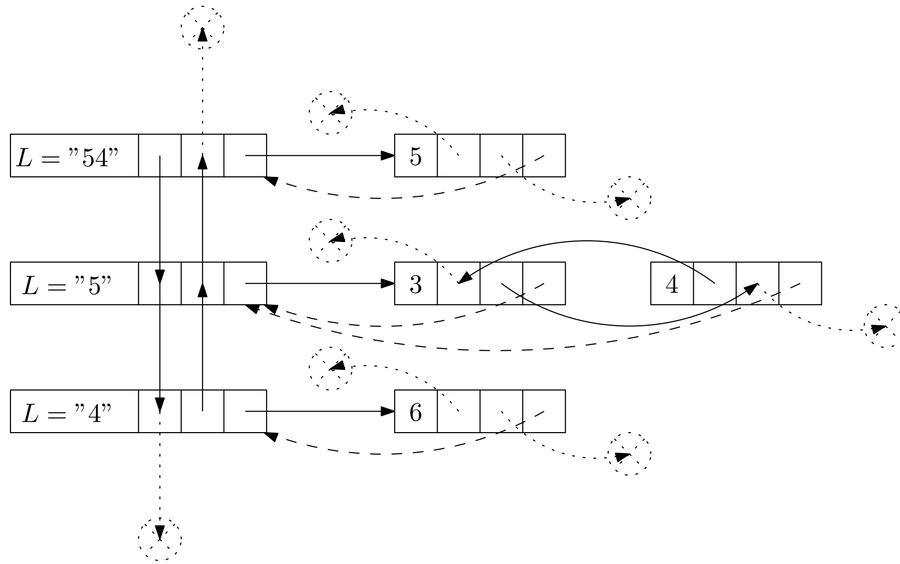


FIGURE 6 – Schéma représentatif d'une partition

On peut alors constater que la complexité temporelle de LexBFS en utilisant cette structure de donnée est bien $O(n+m)$.

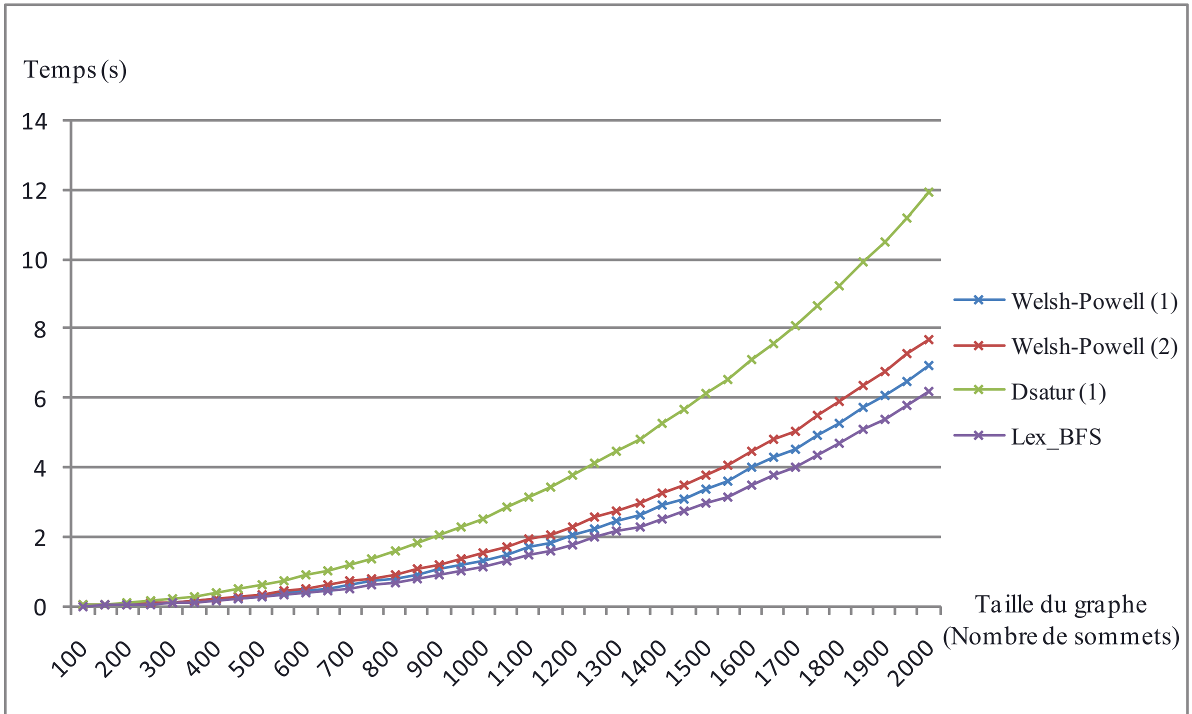
4 Résultats expérimentaux

4.1 Sur des graphes pseudo-aléatoires

Les trois algorithmes présentés ont été implémentés de plusieurs manières en Caml Light : usage de différents tris pour Welsh-Powell, différentes structures de données pour DSATUR, etc. Dans un premier temps nous avons donc essayé ces programmes sur des graphes générés de manière pseudo-aléatoire d'ordres variés, et de densité moyenne $\frac{1}{2}$. Les tableaux qui suivent présentent les temps d'exécution obtenus.

Les tests ont été menés sur un ordinateur fonctionnant sous **Windows XP SP3**, équipé d'un **Intel® Core™ 2 Duo E6400 2.13 GHz** et de **2.00 Go** de RAM.

Figure 7 – Temps d'exécution, de 100 à 2000 sommets



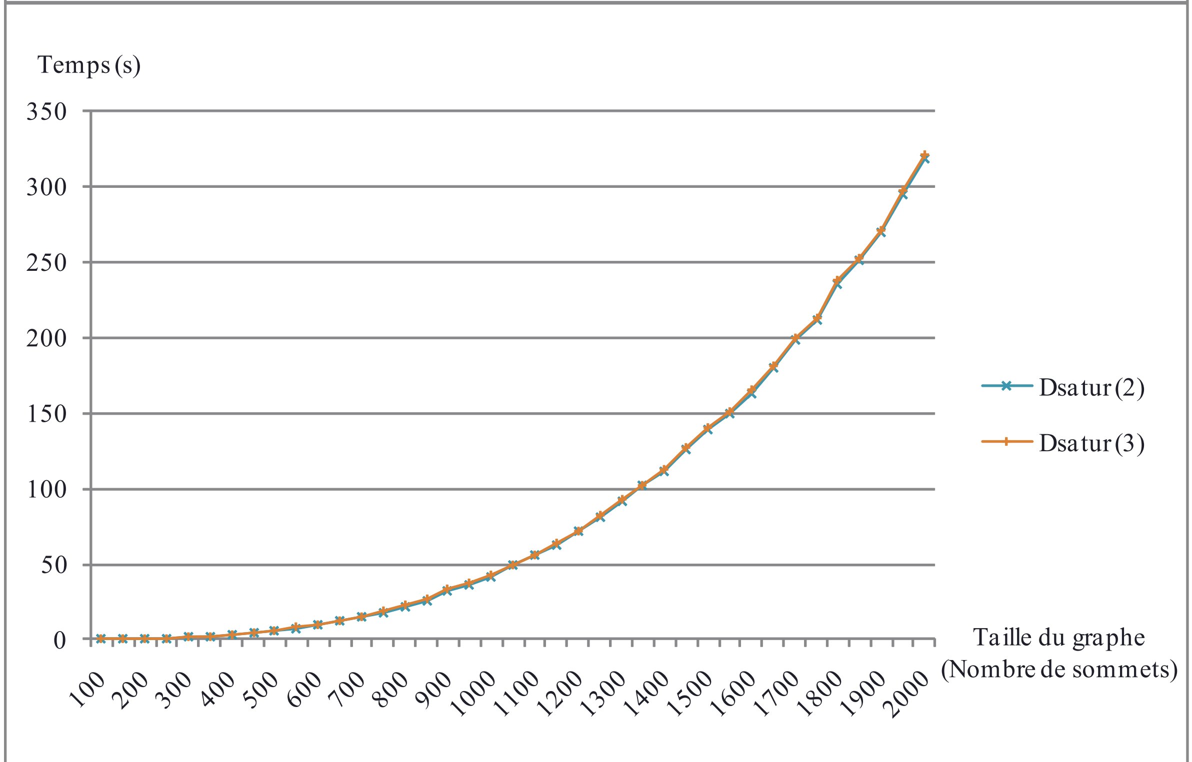
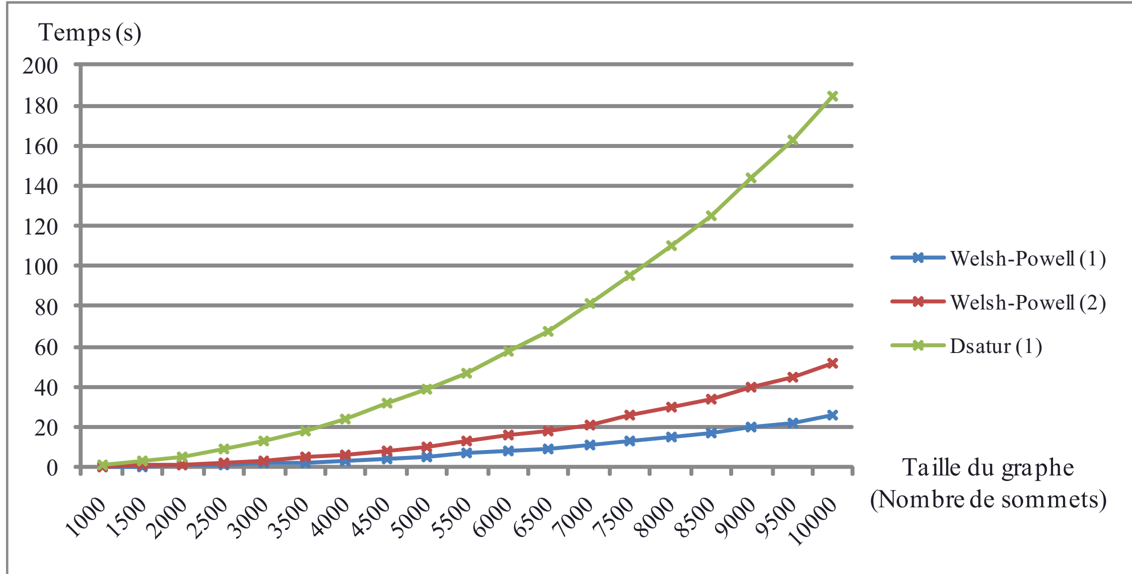


Figure 8 – Temps d'exécution, de 1000 à 10 000 sommets

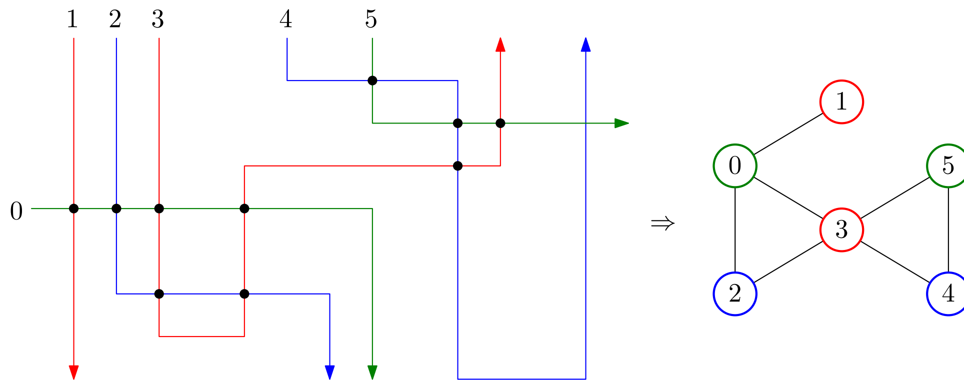


Des problèmes de dépassement de capacité de mémoire n'ont pas permis de mener certains tests jusqu'à 10 000 sommets. Nous avons tenté d'adapter la structure d'affinage de partition pour Dsatur3, mais comme nous pouvons le voir le résultat est décevant : cette structure ne convient manifestement pas à un tel algorithme. Remarquons aussi que le tri par fusion utilisé pour Welsh-Powell1 s'avère plus efficace sur ces graphes que le tri par dénombrement utilisé pour Welsh-Powell2. Globalement nous voyons également que Welsh-Powell et LexBFS sont plus rapides que Dsatur, même si ce dernier offre souvent de meilleurs colorations sur un graphe quelconque.

4.2 Sur des graphes de DIMACS

Afin de mesurer l'efficacité des algorithmes en terme de nombre de couleurs utilisées, nous avons utilisé des graphes publiés par le centre DIMACS (pour *Center for Discrete Mathematics and Theoretical Computer Science*). L'intérêt est double : ces graphes fournissent un benchmark efficace pour tester l'efficacité des colorations sur des problèmes concrets, et ils soulignent les applications multiples du problème de coloration dans la vie réelle : allocation de registre lors de la compilation de code, planification d'horaire, un problème d'échec, configuration de réseaux optiques, problème des carrés latins, etc.

- Réseaux optiques :
  Combien de couleurs différentes faut-il utiliser pour que deux faisceaux de même couleur ne se croisent pas ?



- Graphes de reines :
  Étant donné un échiquier de taille $n \times n$, peut-on placer n ensembles de n reines de même couleur de sorte que deux reines de même couleur ne puissent pas se prendre ?

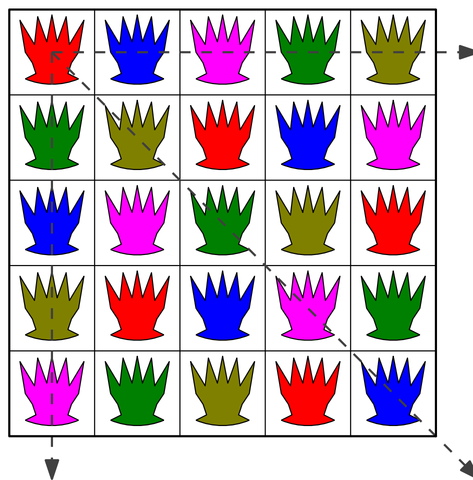


FIGURE 9 – Quelques illustrations des problèmes de DIMACS

TABLE 1 – Comparaison de différents algorithmes sur des graphes de la librairie DIMACS

Graphe	Problème correspondant	Nombre de :			Nb Couleurs		Temps d'exécution (s)				
		Sommets	Arêtes	Coloration	WP	Dsatur	Welsh-Powell		Dsatur		
		n	m	$\chi(G)$			1	2	1	2	3
inithx.i.1.col	Allocation de registre	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	Planification	352	14612	? ? ?	34	27	0	0.015	0.094	0.547	0.047
jean.col	Livre	80	508	10	10	10	0	0	0	0.015	0
queen11_11.col	Échec	121	3960	11	17	15	0	0	0.015	0.11	0.015
myciel4.col	Graphes de Mycielski	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	Réseaux optiques	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	Carrés latins	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

Nous pouvons voir que dans la plupart des cas, WP (pour Welsh & Powell) et DSATUR fournissent la coloration optimale du graphe considéré. Cependant lorsqu'il y a des différences, on voit que 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 pour qg.order40.col), DSATUR fournit bien une meilleure coloration que WP.

Au niveau des temps d'exécution, les graphes présentés étant de faibles tailles (généralement moins de 100 000 arêtes, et presque un million pour qg.order100.col), les temps obtenus sont assez peu élevés et ne présentent pas de grandes différences. Toutefois celles-ci deviennent très marquées pour le dernier exemple ; on peut même s'étonner de voir Dsatur3, qui utilise l'affinage de partition, se distinguer : sans doute les symétries que présente qg.order100.col sont propices à sont exécution.

Conclusion et perspectives

Nous conclurons cette étude en insistant sur les différences d'efficacités constatées, qui dépendent autant de l'implémentation effectuée que des heuristiques considérées. La coloration de graphe est un domaine de recherche très actif et, comme nous avons pu le voir, ses applications sont multiples. Certains problèmes permettant parfois de faire des hypothèses sur la structure du graphe à colorer, on peut envisager de développer des algorithmes polynomiaux de coloration optimale pour ces cas particuliers.

Signalons enfin 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), multi-coloration (allocation d'une liste de couleurs aux sommets), etc.

Références

1. *Cours d'algorithmique des graphes du MPRI*, Michel Habib, 9 janvier 2008
   Chapitre 5 : Parcours en largeur lexicographique
2. Cours de l'université Nice Sophia Antipolis, Frederic Havet, 2006
   Chapitre 5 : Coloration de graphes
3. *Algorithmic Aspects of Vertex Elimination on Graphs*
   Donald J. Rose, Robert Endre Tarjan, George S. Lueker, juin 1976
   SIAM Journal on Computing, Vol. 5, No. 2, p. 266-283
4. DSATUR, page web par Philippe Rolland
   `http://prolland.free.fr/Cours/Cycle2/Maitrise/GraphsTheory/TP/PrgGraphDsat/dsat.html`
5. DIMACS, graphes de benchmark
   `http://mat.gsia.cmu.edu/COLOR02/`