




09 MAI 2019

SYNTHESE PROJET GRAPHS

COLORATION DES GRANDS GRAPHS REELS

ALEXIS GUYOT – ALEXANDRE LABROSSE

UNIVERSITE DE BOURGOGNE
L3 Informatique



Problématique et enjeux

La coloration de graphes est un problème central en théorie des graphes. Dès 1852, De Morgan a commencé à s'intéresser à cette discipline à travers la coloration de cartes. Aujourd'hui, les domaines d'application sont nombreux : optimisation avec contraintes, allocation de registres, réseaux, cryptographie... Plusieurs algorithmes ont alors été créés tels que l'algorithme glouton, DSATUR, BSC... Ces derniers permettent de calculer l'indice chromatique de graphes avec une précision et une complexité plus ou moins intéressantes. L'importance de ce problème a donné lieu à de nombreuses expérimentations, sans réellement réussir à trouver un algorithme optimal permettant de calculer le nombre chromatique exact en un temps raisonnable pour les très grands graphes (problème NP-complet pour plus de 3 couleurs).

L'objectif de ce projet est donc d'étudier le comportement des algorithmes classiques de coloration de grands graphes vus en cours, et en particulier leurs limites. Nous réfléchirons alors à des pistes permettant de les améliorer.

Dans un premier temps, nous allons décrire l'implémentation des algorithmes suivants : algorithme glouton, algorithme de Welsh et Powell et DSATUR et étudier leur complexité. Ensuite, nous comparerons leurs temps d'exécution sur plusieurs exemples, et enfin nous proposerons des pistes de solutions pour améliorer notre travail.

Structures et algorithmes mis en œuvre

```
//const int n = 36692;           //Graphe Email-Enron.txt
//const int n = 54573;           //Graphe HR_edges.csv
const int n = 82168;             //Graphe Slashdot0902.txt

//Variables générales
//int adj[n][n];                 // matrice d'adjacence, représentation du graphe
vector<vector<int>> adj(n,vector<int>(n,0));

//Variables propres à DSATUR
vector<int> couleur(n);          // contient la couleur de chaque sommet
```

Nous avons décidé d'utiliser une matrice d'adjacence pour représenter notre graphe. Ce type de structure présente deux avantages et deux inconvénients. Concernant les avantages, elle est facile à implémenter et regarder si une arête existe entre deux sommets est très rapide. Pour les inconvénients, le parcours peut vite être long ($O(n)$ pour parcourir les voisins, $O(n^2)$ pour un parcours complet) et implémenter une matrice d'adjacence sur des grands graphes est très vite couteux en place. Avec le recul après avoir manipulé quelques grands graphes, nous avons vite remarqué qu'une liste d'incidence aurait certainement été plus judicieuse pour représenter notre graphe, autant pour la place plus minime qu'elle prend en mémoire que pour son temps de parcours de l'ordre du degré maximum du graphe ou de son nombre de sommets. Il s'agit donc d'une idée d'amélioration possible. En plus de cela, un certain nombre d'autres tableaux à une dimension sont utilisés (pour les degrés, les couleurs, le dsat, ...). Un graphe de n sommets prendra en tout $n^2 \cdot 4$ octets dans la mémoire vive, ce qui est très loin d'être négligeable et est incompatible avec de très grands graphes (de l'ordre de plusieurs centaines de milliers de sommets, voire de millions).

```

void chargerGraphe(std::string nomGraphe) {
    int s1,s2;
    std::fstream infile (nomGraphe);
    while(infile >> s1 >> s2)
    {
        adj[s1][s2]=1;
        adj[s2][s1]=1;
    }
}

```

Pour le chargement des graphes en mémoire, nous utilisons les outils de C++ pour lire un fichier. Les graphes sont récupérés sur le site SNAP (<http://snap.stanford.edu/data/index.html>), aux formats CSV ou TXT. Grâce à cette fonction, on peut lire n'importe quel type de graphe, qu'il soit orienté ou non. S'il l'est, le fait qu'on force la matrice d'adjacence à ajouter une arête de part et d'autre de la diagonale aura pour conséquence de le transformer en graphe non-orienté.

```

int DSATUR() {
    int ppc = 1;           //Plus petite couleur
    int ppcmax = ppc;      //Plus grande couleur utilisée (équivalent de
    int compteur = 1;

    while(ResteNonColorie()){
        int x = dsatMax();
        if(x != -1) {
            ppc = plusPetiteCouleur(x);
            if(ppc > ppcmax) { ni[ppc] = compteur; ppcmax = ppc; }
            couleur[x] = ppc;
        }
        compteur++;
        updateDSAT(x);
    }

    return ppcmax;
}

```

Concernant nos algorithmes de coloration, ils suivent globalement tous la même forme. Dans tous les cas, on détermine puis retourne une variable « ppcmax », qui est en réalité la plus grande couleur utilisée par l'algorithme à la fin de la coloration. En d'autres termes, cette variable correspond au nombre chromatique trouvé. Quand cette variable change de valeur (quand une nouvelle couleur est utilisée), on indique dans le tableau Ni le nombre de sommets qui ont déjà été traités avant le changement. Sinon, chaque algorithme a le même but, trouver la plus petite couleur pouvant être associée à un sommet, puis le lui appliquer. La grosse différence entre les 3 algorithmes mis en place est la façon de trouver le sommet à colorier et l'ordre à respecter. Trois algorithmes de coloration ont été mis en place pour ce projet. Le premier est bien évidemment DSATUR, le deuxième est l'algorithme glouton simple et le troisième est l'algorithme glouton séquentiel de Welsh et Powell.

Pour DSATUR, l'algorithme tourne tant qu'il reste des sommets à colorier. A chaque fois, on récupère le sommet qui possède le DSAT maximum puis on lui applique la plus petite couleur possible en fonction de celles déjà données à ses voisins. Une fois cela fait, on met à jour la table DSAT pour prendre en considération les modifications puis on recommence. Au niveau de la complexité, notre implémentation tourne autour de $O(n^6)$, sans compter le temps d'initialisation des différentes structures avant cela. Si on le compte, on tourne autour de $O(n^9)$.

```

for(int i=0; i<n; i++){
    //On applique la plus petite couleur non utilisée par un de :
    ppc = plusPetiteCouleur(i);
    if(ppc > ppcmax) { ni[ppc] = compteur; ppcmax = ppc; }
    couleur[i] = ppc;
    compteur++;
}

```

L'algorithme glouton simple est, comme son nom l'indique si subtilement, le plus simple des trois dans son implémentation. Il parcourt simplement l'ensemble des sommets dans l'ordre croissant et applique à chaque fois la plus petite couleur possible. Au niveau de la complexité, on tourne pour cet algorithme autour de $O(n^2)$, ce qui est plutôt bon en termes de temps d'exécution.

```

ordonnerDegres();
for(int i=0; i<n; i++){
    x = degresOrd[i][1];
    //On applique la plus petite couleur non utilisée par un de :
    ppc = plusPetiteCouleur(x);
    if(ppc > ppcmax) { ni[ppc] = compteur; ppcmax = ppc; }
    couleur[x] = ppc;
    compteur++;
}

```

L'algorithme glouton séquentiel est une sorte de juste milieu entre les deux précédents. La différence avec le simple réside dans l'ordre de parcours des sommets. Au lieu de les prendre dans l'ordre croissant, on les prend dans l'ordre décroissant selon la valeur de leur degré. Concernant la complexité, cet algorithme se trouve entre $O(n^3)$ et $O(n^5)$. Cette variation est due à l'utilisation d'un tri à bulles dont la complexité varie entre $O(1)$, dans le meilleur des cas, et $O(n^2)$, dans le pire.

```

void updateDSAT(int x){
    vector<int> couleurs;
    vector<int>::iterator it;
    int valeur[1];

    //On détermine la plus grande couleur des voisins
    for(int j=0;j<n;j++){ if(adj[x][j]) {
        //Pour chaque voisin du sommet
        couleurs.clear();
        for(int v=0;v<n;v++){ if(adj[j][v]) {
            //On regarde le nombre de couleurs interdites chez ses voisins
            valeur[0] = couleur[v];
            if(valeur[0] != 0){
                //Si on croise une nouvelle couleur (autre que 0), on l'ajoute à la
                it = std::search(couleurs.begin(),couleurs.end(),valeur,valeur+1);
                if(it==couleurs.end()) { couleurs.push_back(couleur[v]); }
            }
        }
    }}
    if(couleur[j] == 0) { dsat[j] = couleurs.size(); }
}
}

```

Concernant les autres algorithmes du projet, qui sont essentiellement des fonctions d'initialisation, d'affichage ou intermédiaires pour les trois fonctions précédentes, nous avons décidé de seulement détailler celle qui met à jour le tableau DSAT après chaque coloration de sommet avec DSATUR. Pour le principe, on vérifie pour chaque voisin du sommet colorié les couleurs de ses propres voisins. Lors de ce parcours, on compte le nombre de couleurs interdites par rapport à son entourage, en stockant sans doublon les différentes couleurs rencontrées. Pour chaque voisin du sommet colorié,

on met ensuite à jour le champ du tableau DSATUR qui lui correspond avec ce nombre de couleurs interdites. (PS : le code souligné en rouge sur la capture d'écran est une erreur d'affichage de l'IDE Netbeans que nous avons utilisé, elle n'empêche pas la compilation ou l'exécution)

Résultats et interprétations

Exemple de résultat

```
----- DSATUR
Taille du graphe : 4039
Nombre chromatique : 72
Ratio p : 1968/4039
```

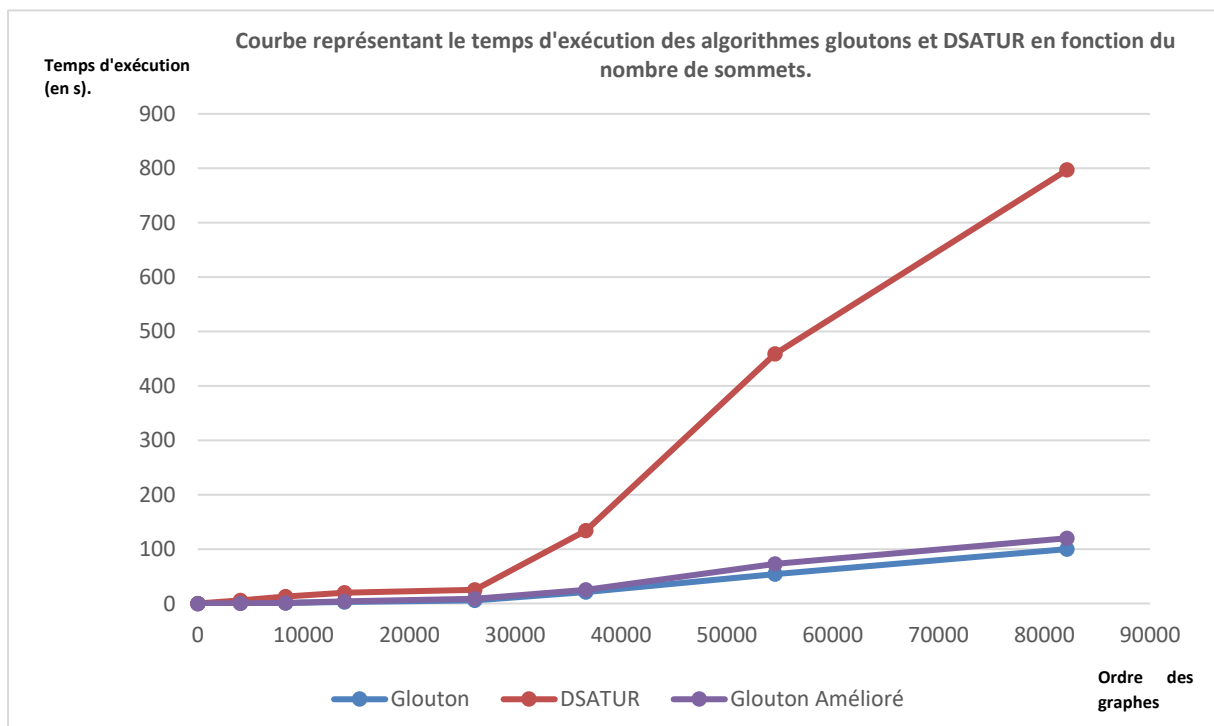
```
----- Glouton
Taille du graphe : 4039
Nombre chromatique : 86
Ratio p : 2656/4039
```

```
----- Glouton Amélioré
Taille du graphe : 4039
Nombre chromatique : 76
Ratio p : 30/577
```

A la fin de l'exécution de chaque algorithme, on affiche quelques statistiques concernant le graphe et la coloration, à savoir l'ordre de ce dernier, le nombre chromatique déterminé par l'algo et le ratio p demandé dans le sujet. Ce ratio permet d'observer le moment dans l'exécution où l'algorithme a trouvé le nombre chromatique du graphe. Nous allons exploiter ces résultats dans la partie suivante.

Temps d'exécution des graphes

Afin de comparer la complexité des deux algorithmes décrits précédemment, nous avons effectué un graphique mettant en relation le temps d'exécution de ces derniers en fonction de leur ordre (nombre de sommets). Voici les résultats que nous obtenons :



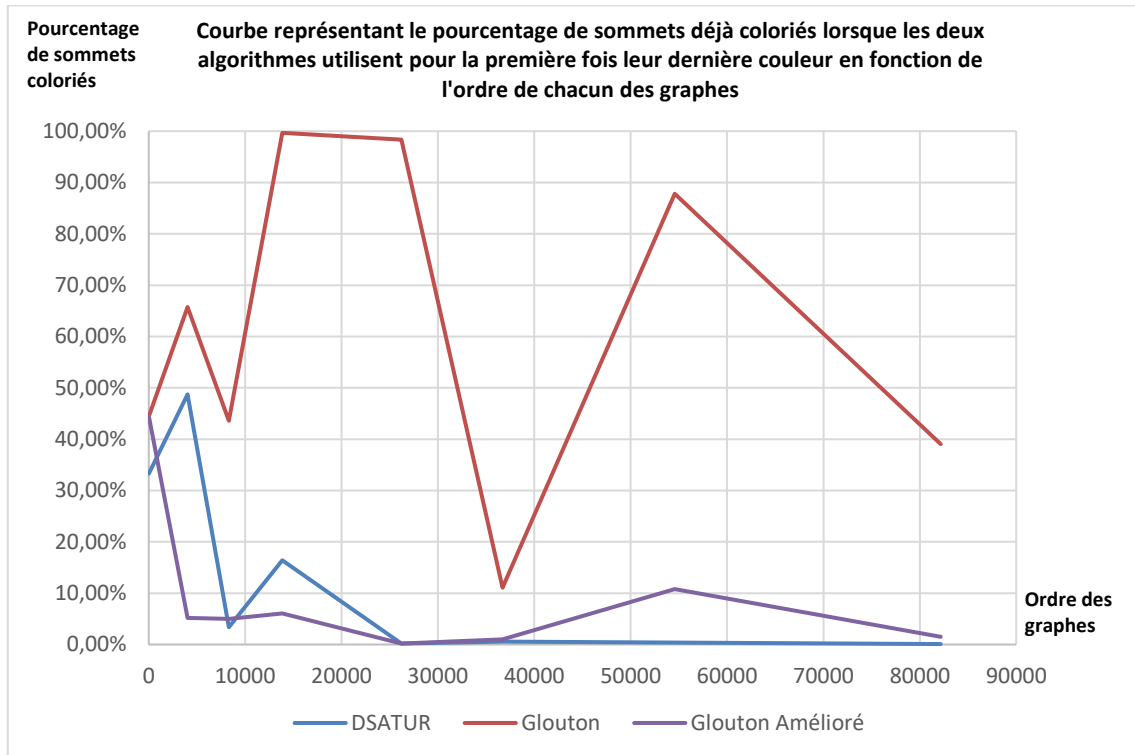
Dans un premier temps, nous pouvons remarquer que pour les graphes composés de moins de 30000 sommets, les algorithmes glouton, glouton amélioré et DSATUR ont un temps d'exécution similaire. C'est lorsque les graphes dépassent ce seuil que leur temps d'exécution diffère. En effet, nous pouvons voir que lorsque les graphes atteignent plus de 80000 sommets, le temps d'exécution de DSATUR est quasiment huit fois supérieur aux temps des deux autres. Ceci semble logique au regard de la complexité des algorithmes calculée précédemment. L'algorithme DSATUR reste cependant le plus intéressant, puisque c'est celui qui donne une estimation la plus proche du véritable nombre chromatique d'un graphe. Cependant, l'algorithme glouton amélioré en donne une estimation assez satisfaisante par rapport à son temps d'exécution. En effet, nous pouvons voir dans le tableau ci-dessous qu'il est assez proche de DSATUR :

Ordre	Nombre chromatique		
	DSATUR	Glouton	Glouton Amélioré
9	3	3	3
4039	72	86	76
8298	24	38	28
13866	30	32	31
26197	44	44	44
36692	25	35	29
54573	15	21	17
82168	32	44	40

Nous n'avons pas pu appliquer ces deux algorithmes sur des graphes d'ordres supérieurs à 82168. En effet, le stockage de ces derniers, sous la forme d'une matrice d'adjacence, oblige notre machine de disposer d'une mémoire très importante. C'est un des inconvénients du stockage de graphe sous cette forme.

Améliorations de l'algorithme DSATUR

Afin d'améliorer l'algorithme DSATUR, notamment en abaissant sa durée d'exécution pour les très grands graphes, nous avons cherché à connaître le pourcentage de sommets déjà coloriés lorsque DSATUR utilise pour la première fois sa dernière couleur (correspondant donc au nombre chromatique qu'il aura calculé). Ce chiffre nous aurait alors permis de pouvoir émettre des hypothèses quant à la nécessité de colorier un graphe dans son intégralité pour trouver son nombre chromatique. Nous avons établi le graphique ci-dessous afin de représenter ce pourcentage en fonction du nombre de sommets du graphe :



Dans un premier temps, nous pouvons constater que l'algorithme glouton utilise sa dernière couleur assez tard dans la coloration de graphe. Il va donc être difficile d'améliorer l'algorithme grâce au pourcentage décrit précédemment. Cependant, nous remarquons que la valeur du ratio ne dépasse jamais $1/3$ pour l'algorithme DSATUR. Cette information est intéressante puisque l'on peut supposer qu'il suffirait de colorier un tiers des sommets d'un graphe afin d'obtenir le nombre chromatique de ce dernier. Cela réduirait ainsi considérablement la durée d'exécution de l'algorithme. Cependant, cela ne permet toujours pas de résoudre le problème évoqué récemment concernant le stockage de la matrice d'adjacence, très gourmande en mémoire. Une solution semble possible : la représentation du graphe sous la forme d'une liste d'incidence. Cette dernière stocke le graphe en listant pour chacun des sommets du graphe les voisins respectifs. Cette forme de structure réduit considérablement la taille de la mémoire requise pour le stockage du graphe, mais est plus compliquée à mettre en place.