Algorithmique : Théorie des graphes

Projet algorithmique

Schleusner Shan

Université Libre de Bruxelles
Faculté des Sciences
Département d'informatique
Avril 2022

Table des matières

1 Introduction

Ce travail a comme objet d'étude l'élaboration de différents algorithmes performant pour obtenir des réponses à différentes questions gravitant autour de la théorie des graphes.

Dans le cadre de ces recherches, nous avons été amené à échanger avec Bappi RAHAMAN et Hugo LEFEBVRE. Nous les remercions pour l'aide apportée et les débats intéressants qui se sont ensuivis.

Afin de construire des graphes facilement utilisables par l'utilisateur, nous avons décidé de réutiliser les classes de la bibliothèque Java algs4.jar dans l'ensemble des implémentations des algorithmes en Java qui ont été présentés. Les tests effectués sur ordinateur ont été faits dans Système d'exploitation Windows de 64 bits et processeur x64 sur un Yoga Slim 7 Pro 14ACH5 OD.

Le squelette de ce rapport est consistué de quatre grandes différentes parties. Dans la première section, nous présenterons un algorithme permettant de donner dans un temps optimal la dégénérescence d'un graphe. Dans la seconde section, nous décrirerons comment obtenir la profondeur d'un sommet fourni en paramètre pour un graphe. Dans la troisième section, il sera question de déterminer une coloration adéquate et de déterminer le lien entre dégénérescence et coloriage. Enfin, nous présenterons l'ensemble des expériences sur des graphes issues de données réelles dans la dernière section importante du projet. Ce rapport se terminera par une conclusion suivie de la bibliographie utilisée.

2 Partie 1: Algorithme de dégénérescence d'un graphe

2.1 Description de l'algorithme

La dégénérescence d'un graphe non-dirigé G est définie comme étant le plus petit nombre k tel que le graphe est k-dégénéré. Un graphe est k-dégénéré si tout sous-graphe du graphe G possède un sommet de degrée k ou moins.

Il est possible d'établir un algorithme permettant de trouver la dégénérescence d'un graphe en un temps linéaire. Un tel algorithme a été proposé par David Matula.

Le processus consiste à "éplucher" le graphe en enlevant en premier lieu les sommets de degréé les plus faibles ainsi que leurs arêtes leur étant éventuellement associés. Par cette procédure, le degrée des sommets dans le graphe peut évoluer au cours de l'algorithme au fur et à mesure que des sommets sont retirés. Il est par conséquent important de réattribuer dynamiquement les informations cohérentes relatives au nombre de voisins des sommets au cours de la procédure d'épluchage.

Par cet algorithme, les sommets ayant un nombre de voisins 0, 1,... seront tour à tour enlevés de l'algorithme lors des opérations p_0, p_1, ... jusqu'à atteindre un graphe vide à l'opération p_k. La dégénérescence du graphe est le nombre k atteint.

En effet, à la fin d'une procédure p, un sommet hypothétique de degrée p+1 doit être nécessairement lié à d'autres sommets de degrée d'au moins p+1. Dans le cas contraire, le voisin d'un degrée inférieur aurait été retiré par la procédure p, le sommet p+1 aurait vu son degrée rétrogradé à p et dès lors, aurait été épluché à son tour par l'algorithme. Par conséquent, il est possible de conclure que tous les sommets du sous-graphe ayant été engendré à la fin d'une procédure p possède un degréé d'au moins p+1. Dès lors, par définition, k doit être supérieur ou égal à p+1.

Continuer cette procédure jusqu'à atteindre un graphe vide permet de plafonner en assurant qu'il n'existe aucun sous-graphe du graphe G ayant l'ensemble des degrées des sommets supérieures à l'index de la procédure finale. Par conséquent, le k trouvé est le k minimum définissant le caractère k-dégénéré du graphe. La dégénérescence du graphe est dès lors effectivement situé à k.

Au cours de ces recherches, plusieurs tentatives de mise en place de cet algorithme en terme de code ont été établies. Il n'a été gardé dans cette section que la version la plus optimale en temps d'exécution. L'ensemble des recherches est détaillée dans une section ultérieure dans la partie 4.

L'algorithme implémenté dans le cadre de ces recherches été imaginé par Batagelj et Zaversnik. Il part du principe détaillé ci-dessus pour obtenir les k-coeurs (cette notion est vue plus en détail dans la section suivante) et en déduire la dégénéresence. L'idée est de manipuler une liste de sommets initiallement ordonnée par degrée pour la transformer en une liste de sommets ordonnée par ordre croissant de k-coeurs par la méthode d'épluchage sommet par sommet.

Au départ, il y a donc d'une part une liste des degrées par sommet et une liste de sommets en ordre croissant de degrée. Pour chaque sommet dans l'ordre, l'ensemble des voisins sont itérés. Si le voisin possède un degrée inférieur ou égal au sommet actuel, c'est qu'il appartient à un k-coeur inférieur égal au courant (voir plus en détail la section consacré au k-coeur). Pour ces types de sommets, il ne nécessite pas d'ordonnancement supplémentaire. Toutefois, si un des voisins à un degré supérieur, son degréé est mis-à-jour et par conséquent sa place dans la liste des sommets est également ajustée.

Par cette procédure, la liste des degrées se met progressivement à jour. Il en résulte que seuls les sommets d'une couche supérieure (k-coeurs) sont étudiés par l'algorithme (les autres pouvant être considérés comme épluchés). A la fin de l'algorithme, lorsque l'ensemble des sommets a été étudiés, le degréé final de l'ultime sommet de la liste des degrées est par conséquent également la dégénérescence du graphe.

2.2 Bornes inférieures et complexités

L'algorithme décrit ci-dessus va retirer tour à tour l'ensemble des sommets du graphe afin de trouver le k minimum. Par conséquent, la complexité de l'algorithme est proportionnelle au nombre de sommets et d'arêtes du graphe. Si le graphe possède v sommets et e arrètes, il s'exécutera en complexité O(max(v, e)). La complexité de cette algorithme est par conséquent linéaire à la taille du graphe.

Seules deux opérations au sein de l'algorithme nécessetient une complexité en O(max(v, e)). La première consiste à obtenir le vecteur qui enregistre le nombre de degrées par sommet. En effet, celui-ci itère chaque sommet et par sommet, il décompte le nombre de degrée. La seconde opération nécessitant une opération de O(max(v,e)) prend place lors du processus d'épluchage où l'ensemble des voisins du sommet sont itérés pour diminuer leur degrée de connexion avec le sommet passé en revue. Cela est précisé en détail dans l'ouvrage "An O(m) Algorithm for Cores Decomposition of Networks".

Il s'agit de la complexité minimale qu'il a été possible d'atteindre dans le cadre de ce projet. D'autres algorithmes, présentés plus en détail dans la section 4, ont été tentés mais disposant de résultats moins efficace en terme de temps d'exécution. Ainsi, cet algorithme met en moyenne 4 secondes pour obtenir la dégénérescence d'un graphe a un million de sommets. Les autres algorithmes ont pris entre dix minutes et plusieurs heures pour étudier un algorithme de cette taille. Cette partie sera détaillée plus en profondeur dans la section 4.

2.3 Implémentation en Java

Dans cette partie, il sera montré comment est implémenté en Java l'algorithme permettant de trouver la dégénerescence d'un graphe fourni en paramètre. L'implémentation de cet algorithme est une version adaptée de celui de Batageli et Zavernsik.

Comme précisé ci-dessus, le but est d'arriver à une liste de sommets ordonnée par degrées ainsi qu'une liste des sommets par degrée pour pouvoir effectuer les transformations nécessaires afin d'arriver à avoir en output, une liste de sommets par degrée. L'ultime degrée étant la dégénéresence.

L'algorithme est divisé en deux parties différentes. Une partie sert à l'initialisation des différentes listes qui vont permettre dans un deuxième temps à élaborer l'algorithme.

Dans un premier temps quatre vecteurs différents sont initialisés.

- Un vecteur "listeCores". Ce vecteur enregistrera dans un premier temps l'ensemble des sommmets classé par ordre croissant des données. Après l'algorithme, il contiendra l'ensemble des sommets par ordre croissant de cores.
- Un vecteur "emplacementSommets". Ce vecteur contient l'index de l'emplacement des Sommets au sein du vecteur listeCore. Il est primordial afin d'obtenir une recherche en temps constant.
- Un vecteur "nbrDegree". Il inventorie tour à tour le nombre de sommets de degrée x ou l'index de délimitations de sommets de même degrée au sein du vecteur "listeCores". Sa taille est donc égale au nombre maximale de degrée du graphe
- Un vecteur "degreeParSommet" qui répertorie le degrée de chaque sommet. A la fin de la procédure, ce vecteur répertorie à quel core appartient chaque sommet. Par conséquent, c'est ce vecteur qui permet de répertorier la dégénérescence.

La phase d'initilisation du graphe consiste en cinq étapes différentes.

- Encoder au sein du vecteur "degreeParSommet" le degrée de chaque sommet. Le degrée maximum du graphe est également déterminée. [Complexité O(max(v,e)]
- Encoder le nombre de sommets qu'il y a pour chacun des degrées et placer ce nombre dans le vecteur nbrDegree. [Complexité O(v)]
- Calculer par le biais de l'étape précédente l'index marquant le changement à des sommets de degrée supérieurs et placer ce résultat dans nbrDegrée.
- Placer dans "ListeCore" et "emplacementSommets" le sommet à sa place déterminée pour arriver à ce que "ListeCore" donne une liste des sommets par ordre croissant et emplacementSommets la position du sommet au sein du vecteur ListeCore. Cette opération peut se faire grâce au vecteur nbrDegree. [Complexité O(v)]
- Remettre à niveau le vecteur nbrDegree qui a été modifié.

Un exemple d'implémentation de l'algorithme en java permettant cette initialisation est affichée ci-dessous

```
for (int sommet = 0; sommet < graphe.V(); sommet++) {
        listeCores.add(-1);
        emplacementSommets.add(-1);
        degreeParSommet.add(graphe.degree(sommet));
        if (graphe.degree(sommet) > degreeMax) {
            degreeMax = graphe.degree(sommet);
        }
    }

    for(int i = 0; i <= degreeMax; i++) {nbrDegree.add(0);}

    for(int j = 0; j < graphe.V(); j++) {
        int degAchanger = graphe.degree(j);
        nbrDegree.set(degAchanger, nbrDegree.get(degAchanger) + 1);}


    int start = 0;
    int num;
    for(int d = 0; d <= degreeMax; d++) {
        num = nbrDegree.get(d);
        nbrDegree.set(d, start);
        start=num+start;
    }


    for(int sommet = 0; sommet < graphe.V(); sommet++) {
        emplacementSommets.set(sommet,
            nbrDegree.get(graphe.degree(sommet)));
        listeCores.set(emplacementSommets.get(sommet), sommet);
        nbrDegree.set(graphe.degree(sommet),
            (nbrDegree.get(graphe.degree(sommet)) + 1));
    }

    for(int d = degreeMax; d > 1; d--) {
        nbrDegree.set(d, nbrDegree.get(d-1));
    }
    nbrDegree.set(0, 1);

}
```

L'algorithme pour trouver la dégénérescence du graphe en tant que tel consiste à parcourir le vecteur "listeCores" et à itérer chaque voisin de ses sommets. Comme expliqué dans la section Algorithme, l'idée est de vérifier si le voisin du sommet possède un degrée supérieur. Si c'est le cas, comme notre sommet a été passé en revue ("épluché"), sa liaison avec ce sommet est enlevée ce qui provoque une diminution de son nombre de degrée. Cette procédure est appliquée pour chaque sommet du graphe.

```
    private static void calculeCore(Graph graphe) {

    for(int i = 0; i < graphe.V(); i++) {
        int sommet = listeCores.get(i);

        for(int voisin: graphe.adj(sommet)) {
            if(degreeParSommet.get(voisin) >
                degreeParSommet.get(sommet)) {

                int degreeVoisin = degreeParSommet.get(voisin);
                int posVoisin = emplacementSommets.get(voisin);
                int posCoreMAJ = nbrDegree.get(degreeVoisin);
                int autreSommet = listeCores.get(posCoreMAJ);

                if (voisin != autreSommet) {
                 Collections.swap(emplacementSommets, voisin,
                     autreSommet);
                Collections.swap(listeCores, posVoisin, posCoreMAJ);
                }
                nbrDegree.set(degreeVoisin,
                    (nbrDegree.get(degreeVoisin)+1));
                degreeParSommet.set(voisin,
                    degreeParSommet.get(voisin)-1);
            }
        }
    }

}
```

Il suffit de sélectionner le dernier sommet du degreeParSommet et de vérifier le degrée obtenu en fin d'algorithme pour obtenir sa dégénérescence.

3 Partie 2: Dégénérescence et coeurs

3.1 Algorithme déterminant la profondeur de chaque sommet

Comme précisé dans l'énoncé, le k-coeurs d'un graphe sont les composantes connexes du graphe après avoir supprimé itérativement les sommets de degrée inférieur à k. L'objectif de l'algorithme est de donner pour chaque sommet du graphe G sa profondeur c(v). La profondeur d'un sommet est définie comme étant comme le plus grand nombre k tel que le sommet v appartient à un k-coeur.

L'algorithme permettant de déterminer les coeurs d'un graphe est similaire à celui déterminant la dégénérescence du graphe.

Il consiste à supprimer itérativement les sommets qui ont un nombre de liaisons inférieurs à un nombre k qui croît au fil de l'algorithme. Par cette tâche, par définition, le graphe est épluché par k-coeur. A l'instar de l'algorithme pour déterminer la dégénerescence d'un graphe, le nombre de liaisons évolue dynamiquement dans le graphe au fil des suppressions de sommets.

Lors de la suppression de ces sommets, un k est attribué dans une liste de sortie. Comme l'algorithme balaye successivement les différentes k couches du graphe, le k trouvé est le k maximal tel que le sommet v appartienne à cet ensemble de k-coeur. Il correspond par définition à la profondeur du sommet.

L'implémentation de l'algorithme part d'une liste de sommets ordonnée de manière croissante par degrée. Chaque sommet est visité tour à tour ("épluché") et les voisins du sommet sont itérés afin d'ajuster éventuellement leur degrée. Seuls les degrés supérieurs sont encore une fois examinés. En effet, les sommets de degrées inférieurs ont déjà été traités d'une manière ou d'une autre par l'algorithme. Les sommets de degrées similaires se trouvent quant à eux déjà dans la bonne range de k-coeur en conséquence du fait que leur degrée aurait été préalablement ajusté à un autre coeur inférieur si ce n'était pas le cas. De ce fait, seuls les sommets de degrées supérieures doivent voir leur liaison se faire réattribués.

A l'instar de l'algorithme pour calculer la dégénérescence, l'objectif ici est de replacer correctement les voisins de degrée supérieur au sommet épluché au sein du vecteur listeCores et de définir leur nouveau degrée une fois la liaison coupée. La fin de l'algorithme voit donc un vecteur trié par core avec un autre vecteur listant le numéro du coeur pour chaque sommet. Lier cette double information permet de donner à l'utilisateur la profondeur d'un sommet au sein du graphe.

3.2 Borne inférieure et complexités

A l'instar de l'algorithme permettant de calculer la dégénerescende d'un graphe, la complexité de cet algorithme peut être estimé en O(max(V, E)) où V est le sommet et E le nombre d'arrètes. Passer en revue l'ensemble des voisins par sommet est l'opération la plus couteuse, que ce soit pour initialiser le vecteur de degrée par sommet ou pour ajuster dynamiquement les voisins ou le vecteur. Toutes les autres opérations peuvent se faire en temps constant ou linéaire au nombre de sommet du graphe.

Puisque l'algorithme est sensiblement le même que celui pour trouver la dégénérescence, il s'agit également de l'algorithme ayant permis d'obtenir les meilleurs résultats parmi tous ceux ayant été tentés (de l'ordre de quelques secondes pour un million de degrée). L'ensemble des recherches antérieures est expliquée dans la dernière section de ce projet.

3.3 Implémentation en Java

L'implémentation est sensiblement la même que la section précédente, en particulier, la partie d'initialisation ne sera en particulier pas réécrite. Il sera mis accent ici sur la manière dont est effectué le déplacement des voisins et la manière de récupérer la profondeur pour un sommet en particulier. Il est par conséquent supposé que l'algorithme dispose d'un vecteur listeCores listant les sommets par ordre croissant, un vecteur donnant la position d'un sommet en particulier dans listeCores, un vecteur du nombre de degrées par sommet ainsi que d'un vecteur donnant la position de la délimitation des coeurs au sein du vecteur listeCores. Dans la suite, le voisin ayant un sommet supérieur au sommet courant sera appelé w.

Pour chaque w, il est nécessaire de mettre à jour son emplacement dans listeCores ainsi que son degréee. A cet effet, l'algorithme va effectuer un double swap entre w et le premier sommet trouvé de l'ancien ayant le même degrée que w avant sa réattribution, sommet qui sera dénommé x. Le premier swap consiste à échanger l'index de l'emplacement de w avec l'index de l'emplacement de x au sein du vecteur emplacementSommets. Le second swap consiste à échanger l'emplacement des deux sommets au sein du vecteur listeCores.

A la fin de la procédure, w se trouve maintenant rangé dans une autre range de degrée. Par conséquent, nbrDegree prend en compte que ce sommet n'appartient plus à sa range et décale d'une unité vers la droite. Cela étant fait, le degrée de w peut finalement être modifié dans le vecteur degreeParSommet.

```
    private static void calculeCore(Graph graphe) {

    for(int i = 0; i < graphe.V(); i++) {
        int sommet = listeCores.get(i);

        for(int voisin: graphe.adj(sommet)) {
            if(degreeParSommet.get(voisin) >
                degreeParSommet.get(sommet)) {

                int degreeVoisin = degreeParSommet.get(voisin);
                int posVoisin = emplacementSommets.get(voisin);
                int posCoreMAJ = nbrDegree.get(degreeVoisin);
                int autreSommet = listeCores.get(posCoreMAJ);

                if (voisin != autreSommet) {
                 Collections.swap(emplacementSommets, voisin,
                     autreSommet);
                Collections.swap(listeCores, posVoisin, posCoreMAJ);
                }
                nbrDegree.set(degreeVoisin,
                    (nbrDegree.get(degreeVoisin)+1));
                degreeParSommet.set(voisin,
                    degreeParSommet.get(voisin)-1);
            }
        }
    }

}
```

Obtenir la profondeur d'un sommet en particulier ne nécessite dès lors que l'entrecroisement des différentes informations disponibles entre listeCores, emplacementSommets et degreeParSommet.

```
private static int getProfondeur(Graph G, int sommet) {
    return
        degreeParSommet.get(listeCores.get(emplacementSommets.get(sommet)));
}
```

4 Partie 3: Coloration d'un graphe

4.1 Démonstration du Lemme

Le nombre chromatique d'un graphe k-dégénéré est inférieur ou égal à k + 1. Il est possible de prouver cette propriété par récurrence. L'idée de cette récurrence est inspiré par le théorème des six couleurs pour les graphes planaires. Une démonstration de ce lemme a également été trouvé

La propriété se prouve facilement pour un graphe comportant un nombre de sommets inférieur à la dégénerescence. Il peut toujours être colorié en moins de k couleurs.

Supposons que la propriété est vrai pour un nombre de sommets v égal à n - 1 et montrons que cette dernière est vrai pour v = n.

Un graphe est k-dégénéré si chaque sous-graphe induit dispose d'un sommet de degrée d'au plus k. Par conséquent, le graphe G dispose lui-aussi d'un sommet v de degrée d'au plus k. Enlevons maintenant ce sommet du graphe et on obtient G' = G - v.

G' est également k-dégénéré puisqu'il dispose lui-même d'un sommet de degrée d'au plus k. Par supposition, G' est donc coloriable au plus en k+1 couleurs. Rajoutons maintenant le sommet v à G'. Il suffit de démontrer que colorier cet ultime sommet ne demande pas plus que k+1 couleurs. Le sommet v enlevé disposait d'un degrée d'au plus k. Par conséquent, il dispose d'au plus k voisins ayant été coloré, ce qui signifie que k+1 couleurs sont suffisants pour colorier ses voisins et le sommet en elle-même. Cela veut donc dire qu'il ne faut que k+1 couleurs pour colorier l'entierté du graphe G k-dégénéré comportant n sommets.

L'hypothèse et la thèse ayant été démontré, cela démontre le théorème.

4.2 Coloration propre d'un graphe

La coloration propre d'un graphe en un nombre minimal de couleur tel que deux sommets adjacents doivent être coloriés en deux couleurs adjacents est un problème NP-complet pour lequel aucun algorithme efficace n'a pu être élaboré à ce jour.

Il existe toutefois un algorithme permettant de trouver une solution acceptable en temps linéaire. Le défaut de cet algorithme étant qu'il ne permet pas toujours de trouver le nombre minimal de couleur disponible. Il permet néanmoins

de trouver avec un ordonnancement précis de toujours trouver une solution d'au plus d+1 couleurs, d étant la dégénerescence du graphe.

Cet algorithme est appelé un "algorithme gourmand". L'algorithme parcours tour à tour chaque sommet du graphe dans un ordre particulier qui sera détaillé ultérieurement. A chaque fois qu'un sommet est colorié, il lui est attribué la première couleur qui n'est pas utilisé par l'un de ses voisins. Une couleur pouvant être représenté par un nombre (0, 1, 2,...), la première couleur peut être considérée comme étant la plus petite couleur n'étant utilisé par aucun des sommets du voisin.

Comme dit précédemment, cet algorithme ne détermine pas un nombre optimal de couleur pour toute série de graphe. L'ordre dans lesquels les sommets sont examinés influe sur le nombre de couleurs utilisés. Une première borne supérieure de cet algorithme peut déjà être établie en observant que un algorithme de cet ordre utilisera au plus le nombre maximal de degrées + 1. En effet, si le sommet avec de degrée maximum M est colorié en premier, il aura au plus M voisins. M + 1 couleurs sont donc suffisantes pour colorier le graphe.

Dans une logique similaire, colorier d'abord les sommets faisant parti du k-coeur le plus important permet de trouver une limite supérieure à d + 1 couleurs utilisées. En effet, ces sommets sont ceux utilisant le plus de couleurs. Par cette méthode, il est assuré que chaque sommet au moment où il est colorié devra uniquement utilisé une couleur différente pour les k autres sommets dont il est connecté dans son k-coeur, ce qui signifie que k+1 couleurs sont suffisantes pour colorier l'ensemble du graphe.

4.3 Borne inférieures et complexités

En entrée, le graphe prend une liste de sommets triés par coeurs qui prend en complexité un temps de O(max(V, E)) (voir section précédente). Pour attribuer la couleur, l'algorithme gourmand passe en revue l'ensemble des sommets du graphe et vérifie pour chacun de ses voisins la couleur à attribuer. Cette procédure s'exécute au minimum dans une durée proportionnelle à la taille du graphe, c'est-à-dire dans une complexité en O(max(V, E)) où V est le nombre de sommets et E le nombre d'arêtes. Ces procédures sont les plus couteuses en terme d'exécution. Toutes les autres opérations prennent un temps linéaire ou constant.

Il existe d'autres types d'algorithmes permettant d'obtenir un meilleur coloriage du graphe, bien que disposant de capacités plus pauvres en terme de vitesse sur des grands graphes. Ceux-ci sont détaillés dans la thèse de Klaus

Lucas Hübner. Citons notamment des algorithmes comme celui de Dsatur utilisant les degrées ayant la saturation maximum ou des algorithmes génétiques.

4.4 Implémentation de l'algorithme en Java

Nous ne reviendrons pas dans cette partie dans la manière de générer une liste triée de k-coeurs. Cet algorithme peut se retrouver dans la section précédente.

L'utilisation de l'algorithme est divisée en deux fonctions différentes afin de faciliter la compréhension. La première fonction se charge d'itérer les sommets et d'attribuer la couleur au sommet. La seconde donne la couleur à utiliser.

Deux vecteurs sont initialisés avant l'utilisation de la fonction.

- Un vecteur "ordreSommets" lisant les sommets par ordre croissant de cores
- Un vecteur "couleurParSommet" mesurant la taille du graphe en nombre de sommet et initiallement comportant que des couleurs "-1" pour signifier l'absence de couleurs.

La fonction procedureNC se charge d'itérer le vecteur ordreSommets par ordre décroissant afin d'obtenir les sommets ayant le k-coeur le plus élevés en premier. Elle charge enregistre ensuite dans un vecteur listeCouleurVoisin les différentes couleurs des voisins dont une couleur leur a déjà été attribué. Puis, par le biais d'une fonction déterminant la couleur, la couleur optimale est assigné au sommet.

```
private static void procedureNC(Graph G) {
      List <Integer> listeCouleurVoisin = new ArrayList <Integer> ();

      for(int j = ordreSommets.size() - 1; j >= 0; j--) {
          int sommet = ordreSommets.get(j);

          for(int voisin: G.adj(sommet)) {
              int couleur_voisin = couleurParSommet.get(voisin);
              if(( couleur_voisin != -1) &&
                   (listeCouleurVoisin.indexOf(couleur_voisin) == -1)) {
                  listeCouleurVoisin.add(couleur_voisin);
              }
          }

          int notre_couleur = obtenirCouleur(listeCouleurVoisin);
          couleurParSommet.set(sommet, notre_couleur);
          listeCouleurVoisin.clear();
      }

  }
```

La fonction assignant la couleur est assez intuitive. Elle tente de proposer la couleur minimale si cette dernière ne figure pas dans le vecteur listant les couleurs utilisés par les voisins. Dans le cas contraire, elle utilise une autre couleur jusqu'à en trouver une qui convient.

```
  private static int obtenirCouleur(List<Integer> listeCouleurVoisin) {
      int couleur = 0;
      while(listeCouleurVoisin.indexOf(couleur) != -1) {
          couleur++;
      }
      couleurmax = Math.max(couleur, couleurmax);
      return couleur;
  }
```

Couleurmax est le nombre de couleur maximal qui est utilisé dans le cadre de l'algorithme. Ce nombre permet de s'assurer que le nombre de couleurs utilisés ne dépassent jamais la dégénérescence.

5 Recherches sur des graphes

5.1 Efficacité des algorithmes

Au cours des recherches, différents types d'algorithmes ont été utilisés afin de résoudre les problèmes. Nous avons gardé pour l'écriture du rapport que ceux donnant les meilleurs utilisations pour des grands graphes.

Nous avons utilisé des graphes ayant de plusieurs milliers à un million de sommets afin de tester les différentes versions de l'algorithme. De plusieurs heures, nous avons réussi à diminuer le temps d'exécution pour tendre vers quelques secondes pour l'algorithme actuel.

Une version précédente de l'algorithme pour calculer à la fois les cores et la dégénérescence utilisait un triple vecteur:

- Un vecteur "DejaVisites" qui enregistrait les sommets épluchés par l'algorithme.
- Un vecteur "nbrVoisinsParSommet" qui enregistre le nombre de voisins du sommet au fur et à mesure que des liaisons se coupent.
- Un vecteur de vecteur "tableSommetsParDegrée" qui place dans chaque compartiment de degrée d l'ensemble des sommets ayant un degrée de cette mesure

Une implémentation inefficace de cette algorithme est donnée ci-dessous:

```
private static void majLiaisons(int sommet, int degree) {
    int indiceAEnlever =
        tableSommetsParDegree.get(degree+1).indexOf(sommet);
    tableSommetsParDegree.get(degree+1).remove(indiceAEnlever);
    tableSommetsParDegree.get(degree).add(sommet);

}
```

```
private static int trouverDegenerescence(Graph G) {
    int k = 0;
       for(int s = 0; s < tableSommetsParDegree.size(); s++) {
           for(int i = s; i >= 0; i--) {
              while(tableSommetsParDegree.get(i).size() > 0) {
               k = Math.max(k, i);
               int sommet = tableSommetsParDegree.get(i).remove(0);
               DejaVisites.add(sommet);

               for (int w: G.adj(sommet)) {

                   if(0>DejaVisites.indexOf(w)) {
                       int nouveauDegree = nbrVoisinsParSommet.get(w)
                           - 1;
```

```
                        nbrVoisinsParSommet.set(w,
                            Integer.valueOf(nouveauDegree));
                        majLiaisons(w, nouveauDegree);
                    }
                }

            }
        }
    }
    return k;
}
```

Plusieurs problèmes majeurs sont observables dans cet algorithme:

- La recherche dans le vecteur DejaVisites ne se fait pas en temps constant. Elle augmente au fur et à mesure que la liste s'allonge. Cela veut dire concrètement que pour un graphe d'un million de sommets, les derniers sommets devront parcourir une étape proche d'un million de sommets.
- L'algorithme passe un temps considérable à nettoyer les sommets de degrées inférieurs dont il est déjà connu au moment de l'algorithme qu'ils n'auront aucun impact sur le calcul de la dégénérescence.
- La mise à jour du sommet nécessite également de parcourir un compartiment de tableSommetsParDegrée pour chaque voisin du graphe. Un temps d'une complexité d'environ n/d à chaque opération.

L'estimation précise en terme de secondes ou millisecondes du programme d'exécution du programme pour de large graphe n'a pas pu être obtenu avec précisons étant donné du temps considérable qu'il mettait à obtenir un résultat. Nous estimons à plus de trois heures de temps pour parvenir à obtenir la dégénérescence d'un graphe à un million de sommets.

Une amélioration simple permet de descendre l'exécution du programme pour un graphe à un million de sommets de quelques heures à une exécution calculé précisemment en dessous de la barre des 10 minutes (9,55 minutes). Il s'agit de placer les sommets à une place déterminée dans le vecteur DejaVisites afin d'obtenir un temps constant à l'utilisation.

```
(...)
DejaVisites.set(sommet, sommet);

                for (int w: G.adj(sommet)) {

                    if(DejaVisites.get(w) == -1){
(...)
```

Il a également été tenté d'obtenir un temps constant pour la recherche du sommet dans la liste de tableParSommetDegree. Cela engendrait toutefois quelques complexités supplémentaires notamment pour garder trace des sommets enlevés dans la table entre-temps. Il a été décidé à cette étape de passer à l'algorithme décrit plus haut.

Cet algorithme peut être également amélioré de manière plus performante en notamment excluant d'emblée des calculs supplémentaires sur les degrées d'ordre inférieures. Ceci nécessite toutefois un remaniment profond qui n'aboutira au mieux qu'à un calcul similaire à l'algorithme présenté dans le cadre de ce projet.

5.2 Caractéristiques des graphes

Différents types de graphes ont été utilisés pour tester le nombre chromatique, la dégenerescence et la profondeur de sommets ayant de quelques dizaines à un million de sommets.

Au cours de ces recherches, nous avons notamment testés l'algorithme délimitant les k-coeurs sur des arbres, dont nous avons obtenu une coloration optimale. Cette propriété est confirmée par des travaux antérieures dans le domaine de la coloration des graphes, notamment "Classical colorying of graphs" de Kosowki de 2004.

Pour des graphes d'un million de sommets, nous avons obtenu en moyenne les résultats suivants:

- 30 secondes sont nécessaires pour l'initialisation du graphe gérée par la librairie algs4.jar.
- Le calcul de la dégénérescence d'un graphe s'obtient en moyenne en quatre secondes
- Obtenir la profondeur d'un sommet prend un laps de temps similaire.
- Il s'agit également du cas du calcul de la coloration d'un graphe fourni en paramètre.

6 Conclusion

Ce projet nous a apporté un vaste socle de connaissance en ce qui concerne la théorie des graphes et aussi l'algorithmie. Il nous a aidés à manipuler des grands graphes, d'imaginer des solutions permettant de les itérer dans un temps optimal, et d'appronfondir la manière dont sont effectués les recherches en informatique théorique.

Nous avons conceptualisé notre code pour tendre vers une compréhension accrue de la part du lecteur tout en maximisant de manière efficace les différentes étapes pour pouvoir appliquer nos algorithmes sur des graphes importants. Nous avons manipulés au cours de ces projets différents apprentissages relatifs à notre cursus, aussi bien dans le champs des mathématiques, de la recherche scientifique et du code. Il nous a également permis d'augmenter notre compréhension sur les spécifités du langage Java.

Ce projet nous aura demandé de nombreux jours de recherches, parfois quelques nuits, de travail et si le perfectionnement de quelques tâches est toujours possible, nous sommes satisfaits du travail présenté et inspirés pour en apprendre davantage

7 Bibliographie

- Kosowski, Adrian; Manuszewski, Krzysztof (2004), "Classical coloring of graphs", Marek (ed.).
- V. Batagelj, Matjaz Zaversnik, "An O(m) Algorithm for Cores Decomposition of Networks", University of Ljubljana.
- Klaus Lukas Hübner, "Coloring Complex Networks", Institute for Theoretical Computer Science.
- D. W. Matula, L. L. Beck, "Smallest-last Ordering and Clustering and Graph Coloring Algorithms", Communications of ACM