Degenerescence - Coloration - Coeurs

Robin Becker (521513), Noé Bourgeois (496667)

March 2022

Table des matières

1 Introduction

Ce projet consiste en la réalisation et l'étude de 3 tâches relatives aux graphes : le calcul de leur dégénérescence, leur colorisation (et donc le calcul de leur nombre chromatique) et le calcul de la profondeur des sommets. Pour se faire, nous avons implémenté plusieurs algorithmes dans le langage JAVA.

Ce rapport a pour but de présenter et expliquer les différents algorithmes utilisés pour la résolution de ces tâches, de comparer leurs complexités à des bornes inférieures et de montrer les résultats de nos tests de ces algorithmes sur de grands graphes issus de données réelles provenant du site de Stanford University.

2 Dégénérescence

2.1 Introduction

cf.énoncé : Dans ce qui suit, on considérera des graphes non-dirigés, simples et sans boucle. Un sous-graphe d'un tel graphe G est un graphe obtenu en supprimant certains sommets et certaines arêtes de G. Un graphe G est dit k-dégénéré si pour tout sous-graphe de G, y compris G lui-même, il existe un sommet de degré au plus k. La dégénérescence d'un graphe est le plus petit nombre k tel que le graphe est k-dégénéré

https://en.wikipedia.org/wiki/Degeneracy_(graph_theory)

2.2 algorithme

Matula and Beck (1983) outline an algorithm to derive the degeneracy ordering of a graph $G = (V, E)$:

1: $L \leftarrow$init
2: **for** vertex v in G **do**
3: $d_v \leftarrow$adj[v] ▷ Compute a number dv for each vertex v in G, the number of neighbors of v that are not already in L.Initially, these numbers are just the degrees of the vertices.
4: **end for**
5: $D \leftarrow array$
6: **for** vertex v in G **do**
7: $D[i] \leftarrow d_v$ ▷ Initialize an array D such that D[i] contains a list of the vertices v that are not already in L for which dv = i.
8: **end for**
Initialize k to 0.
9: $k \leftarrow 0$
10: **while** D[i] is empty **do**
11: $i \leftarrow +1$ ▷ Scan the array cells D[0], D[1], ... until finding an i for which D[i] is nonempty.
12: $k \leftarrow max(k, i)$ ▷ Set k to max(k,i)
13: $v \leftarrow D[i]$ ▷ Select a vertex v from D[i].
14: $L \leftarrow v$ ▷ Add v to the beginning of L and remove it from D[i].
▷ For each neighbor w of v not already in L : ▷ subtract one from dw ▷ move w to the cell of D corresponding to the new value of dw.
15: **end while**

2.2.1 Description

2.2.2 Complexité

Temps :
- Borne inférieure : $\Omega(|V| + |E|)$
- Pire cas : $\mathcal{O}(|V| + |E|)$

Espace :
- Borne inférieure : $\Omega(|V| + |E|)$
- Pire cas : $2|E| + \mathcal{O}(|V|)$

2.2.3 Mise en oeuvre en Java

La méthode `getDegeneracy()` est inspirée de la librairie JGraphT, classe Coreness.java, méthode `lazyRun()`.

see : https://github.com/jgrapht/jgrapht/blob/master/jgrapht-core/src/main/java/org/jgrapht/alg/scoring/Coreness.java

Lors d'une discussion avec Anton Romanova, nous avons, via l'utilisation de l'outil "IntelliJ IDEA profiler", constaté que l'exécution était ralentie par la méthode iterator().next() appelée à chaque assignation d'un

nouveau sommet à retirer sur les `Set` de sommets de même degré. En remplaçant les `Set` par des `Treeset` dont la structure de données est un arbre rouge-noir et `iterator().next()` par `first()`, nous avons, par exemple, fait passer l'exécution sur un graphe à 1 971 281 nœuds et 5 533 214 arêtes de 18 minutes à 2 secondes.

see : https://www.jetbrains.com/help/idea/run-with-profiler.html

Nous commençons par initialiser la dégénérescence à 0, si le graphe contient 1 ou 2 sommets, c'est la valeur qui sera retournée.
Un dictionnaire des sommets et leur profondeur, nommé "L" dans l'algorithme de Matula and Beck, contiendra les sommets ayant été virtuellement retirés du graphe.
Les graphes traités étant sans boucle, le degré maximum est logiquement initialisé à V-1.
Nous initialisons ensuite degreeIndexedPriorityQueue sous la forme d'une liste de `TreeSet`.
Le degré minimum, inititialisé au nombre de sommets, sera réduit par la suite. Une "carte" des sommets et leur degré sera également d'une grande aide, celle-ci est donc inititialisée sous la forme d'une ... `"Map"`.
Chaque sommet est ensuite stocké dans le TreeSet d'index correspondant à son degré et le degré minimum est reduit au degré du sommet de degré minimum.

```
public static int getDegeneracy(Graph g){
        int degeneracy = 0;
        Map<Integer, Integer> depths = new HashMap<>();
        int maxDegree = g.V() - 1;

        TreeSet[] degreeIndexedPriorityQueue = (TreeSet[]) Array.newInstance(TreeSet.class

        for (int i = 0; i < degreeIndexedPriorityQueue.length; i++) {
            degreeIndexedPriorityQueue[i] = new TreeSet<>();
        }

        int minDegree = g.V();
        Map<Integer, Integer> degreesMap = new HashMap<>();
        for (int v = 0 ; v < g.V() ; v++) {
            int d = g.degree(v);
            degreeIndexedPriorityQueue[d].add(v);
            degreesMap.put(v, d);
            minDegree = Math.min(minDegree, d);
        }
```

Tant que le degré minimum est inférieur à la quantité de sommets, tant qu'un Treeset contenant au moins un sommet n'est pas trouvé, le degré minimum est augmenté de 1.
Le premier sommet de degré minimum trouvé est supprimé du TreeSet, la carte des profondeurs et la dégénérescence sont mises à jours.
Pour chaque sommet adjascent à celui retiré, si son degré est supérieur au degré minimum et qu'il n'a pas déjà été virtuellement retiré, sa position et son degré sont mis à jour. La recherche du sommet de degré minimum recommence alors.

```
        /*
         * Extract from degreeIndexedPriorityQueue
         */
        while (minDegree < g.V()) {
            /// minimum degree is smaller than the number of vertices
            TreeSet<Integer> minDegreeKey = degreeIndexedPriorityQueue[minDegree];
            if (minDegreeKey.isEmpty()) {
                minDegree++;
```

```
                continue;
            }
            int vertex_to_remove = minDegreeKey.first();
            /// remove the vertex from the TreeSet O(log(n))
            minDegreeKey.remove(vertex_to_remove);
            /// put the min degree at hashset vertex_to_remove index  O(1)
            depths.put(vertex_to_remove, minDegree);
            degeneracy = Math.max(degeneracy, minDegree);

            for (int u : g.adj(vertex_to_remove)) {
                int uDegree = degreesMap.get(u);
                /// verify if the minDegree is smaller than the degree of u O(1) and if u
                if (minDegree < uDegree && !depths.containsKey(u)) { // O(1)
                    /// remove the vertex from the TreeSet O(log(n))
                    degreeIndexedPriorityQueue[uDegree].remove(u);
                    uDegree--;
                    //// update u Degree in degreesMap O(1)
                    degreesMap.put(u, uDegree);
                    /// update u position in degreeIndexedPriorityQueue O(log(n))
                    degreeIndexedPriorityQueue[uDegree].add(u);

                    ///update minDegree if necessary O(1)
                    minDegree = Math.min(minDegree, uDegree);
                }
            }
        }
        return degeneracy; // O(V+E)
    }
```

3 Coloration

3.1 Introduction

L'algorithme Wave Function Collapse est à l'origine un des meilleurs algorithmes utilisés pour la génération procédurale de terrain dans les jeux vidéos. Mais ses applications vont de la génération de texture à la résolution de sudoku, et nous avons choisi d'utiliser une de ses variantes : le Wave Function Collapse Coloring (WFC-C) pour la coloration de graphe.

WFC-C est un algorithme gourmand qui est capable de minimiser les conficts de couleurs, et ainsi le besoin d'utiliser des couleurs supplémentaires, pour atteindre un nombre chromatique très proche du minimum possible. Il est en théorie extrêmement plus performant et précis que la plupart des autres algorithmes de coloration.

3.1.1 Description

L'idée derrière la façon dont fonctionne WFC tient ses origines dans la physique quantique, et plus précisément, la superposition quantique. Chaque sommet du graphe possède une superposition de couleurs possibles. Quand la couleur d'un de ces sommets est observée, c'est-à-dire déterminée, l'information se propage aux sommets voisins qui voient leur domaines de couleurs possibles diminuer à leur tour, ce qui rend possible une coloration en cascade.

Il est impossible de connaître à l'avance le domaine des couleurs possibles, étant donné que le nombre chromatique du graphe fait partie de la réponse. Mais nous pouvons l'estimer en utilisant le théorème de Brook qui dit que si un graphe G avec un degré maximum m (m >2) ne contient aucun sous-graphe étant des (m+1)-graphes complets, alors le nombre chromatique de G est au maximum m.
Ainsi, nous pouvons initialiser dans notre algorithme le domaine M des couleurs possibles comme étant le degré maximum du graphe G. Dans la description de l'algorithme qui suit, nous définissons l'entropie d'un sommet comme étant le nombre de couleurs avec lesquelles on ne peut pas colorier ce sommet. Si l'entropie d'un sommet atteint la valeur de M, la coloration est impossible : il faut ajouter une nouvelle couleur et recommencer l'algorithme.

Pour éviter au maximum ce genre de cas, il est nécéssaire de commencer la coloration par le sommet qui a le plus haut degré. La première couleur lui est attribuée et l'information est propagée à ses voisins.

```
public static void initAlgo(Graph G) {
    entropy = new LinkedList[G.V()];
    int origin=0;
    for (int i=0; i<G.V(); i++) {
        int degree = G.degree(i);
        if (degree > M) {
            M = degree;
             origin = i;
        }
        uncolored.add(i);
        entropy[i] = new LinkedList<>();
    }
    colorize(origin, collapse(origin));
    propagate(origin);
}
```

Colorize() colorie un sommet avec le résultat de la fonction collapse() qui renvoie la plus petite couleur disponible dans le domaine du sommet.

```
public static int collapse(int v) {
    for (int i=0; i<M; i++) {
        if (!entropy[v].contains(i)) { return i; }
    }
    return -1;
}
```

Propagate() propage l'information aux sommets environnants. Leur entropie augmente car une couleur de moins leur est accessible. S'ils n'ont plus qu'une couleur disponible, cette couleur leur est attribuée et cette nouvelle information est propagée à son tour.

```
public static void propagate(int v) {
    Stack<Integer> notPropagated = new Stack<>();
    notPropagated.add(v);
    int propagationOrigin;
    while (!notPropagated.isEmpty()) {
        propagationOrigin = notPropagated.pop();
        int colorOrigin = color(propagationOrigin);
        for (int neigh: G.adj(propagationOrigin)) {
            if (!coloration.containsKey(neigh)) {
                if (entropy(neigh) < M-1) { entropy[neigh].add(colorOrigin); }
                if (entropy(neigh) == M-1) {
                    colorize(neigh, collapse(neigh));
                    notPropagated.add(neigh);
                }
            }
        }
    }
}
```

Une fois que la propagation s'est terminée, l'algorithme entre dans sa boucle principale : tant qu'il reste des sommets non colorés, le sommet avec la plus grande entropie est sélectionné via la fonction observe() et est coloré à son tour, entrainant une nouvelle propagation. Le cas où l'on est incapable de colorier un sommet a été expliqué précédemment.

```
public static int observe() {
    int lowestEntropy = -1; int lowestVertex = -1; int curr_entropy;
    for (int vertex: uncolored) {
        curr_entropy = entropy(vertex);
        if (curr_entropy > lowestEntropy) {
            lowestEntropy = curr_entropy;
            lowestVertex = vertex;
        }
    }
    if (lowestEntropy == M) { lowestVertex = -1; }
    return lowestVertex;
}
```

3.1.2 Complexité

La complexité en temps de l'algorithme est O((E+V)*V) et la borne inférieure pour le problème de coloration de graphe est omega $\Omega(E+V), notreimplmentationn'estdoncpasoptimale.$

3.1.3 Preuve du Lemme

Pour vérifier la validité d'une coloration, nous utilisons le Lemme suivant :

Le nombre chromatique d'un graphe k-dégénéré est inférieur ou égal à k+1.

Nous pouvons démontrer ce Lemme par récurrence :

Cas de base (n = 1) : le Graphe G a 1 seul sommet. Le degré de ce sommet est 0 et la dégénérescence k de G est aussi 0. Pour colorier G, il ne faut qu'une seule couleur, donc le nombre chromatique est k+1.

Hypothèse : Si P(n) est vrai, alors P(n+1) est vrai aussi.

Si G est un graphe k-dégénéré à n+1 sommets, alors G contient au moins un sommet v dont le degré est inférieur ou égal à k. Si nous retirons ce sommet, et selon la définition de la dégénérescence, nous obtenons un graphe G-v dont la dégénérescence reste k. Par l'hypothèse P(n), G-v est coloriable en k+1 couleurs.
Si nous rajoutons à nouveau le sommet v à G-v, et puisque le degré de v est au plus k et qu'il a donc au plus k sommets voisins, cela nous laisse avec une couleur de plus à utiliser pour la coloration.
Ainsi nous prouvons par récurrence nothe hypothèse qu'un graphe k-dégénéré est coloriable avec au plus (k+1) couleurs.

4 Dégénérescence et Cœurs

4.1 Introduction

cf. énoncé : Les k-cœurs d'un graphe G sont les composantes connexes du graphe obtenu après avoir supprimé itérativement les sommets de degré inférieur à k. De manière équivalente, il s'agit d'un sous-graphe connexe de G pour lequel tous les sommets sont de degré au moins k et qui est maximal pour l'inclusion, donc auquel on ne peut ajouter aucun autre sommet sans perdre cette propriété. La dégénérescence d'un graphe est le plus grand nombre k tel que le graphe a un k-cœur.
Pour un sommet v de G, on peut définir sa profondeur c(v) comme le plus grand nombre k tel que v appartient à un k-cœur.

4.2 algorithme

D'après 4.1Introduction, l'algorithme de `getDepths()` est le même qu'en 2.2 algorithme

4.2.1 Description

Bonus : getKCoreSizes() retourne les tailles des coeurs.

4.2.2 Complexité

cf. 2.2.2Complexité

4.2.3 Mise en oeuvre en Java

Le code de

```
public static Map<Integer, Integer> getDepths(Graph g)
```

est identique à celui de `getDegeneracy()` à la différence près que la variable `degeneracy` est absente et la valeur de retour est `depths`.

```
public static Integer getVertexDepth(Graph g,
                                      int v)
    {
        getDepths(g);
        return depths.get(v);
    }


public static Map<Integer, Integer> getKCoreSizes(Graph g)
    {
        Map<Integer, Integer> depths = getDepths(g);
        Map<Integer, Integer> k_cores_sizes = new HashMap<>();
        for (int depth : depths.values()) {
            if (k_cores_sizes.containsKey(depth)) {
                k_cores_sizes.put(depth, k_cores_sizes.get(depth) + 1);
            } else {
                k_cores_sizes.put(depth, 1);
            }
        }
        return k_cores_sizes;
    }
```

graphe	nœuds	arêtes	CCM
1v	1	0	1
2v1e	2	1	1
graphtest	4	5	0.6
ego-facebook	4039	88234	0.6055
roadNet-PA	1088092	1541898	0.046
roadNet-CA	1971281	5533214	0.0464
com-LiveJournal	4036538	34681189	0.2843
com-friendster	124833781	1806067135	0.1623

Table 1 – Données

graphe	dégénérescence	TMCD	colorabilité	TMCC
1v	0	0.7719	1	0,5106
2v1e	1	0.7013	2	0,4722
graphtest	2	0.9016	3	2,828
ego-facebook	115	80	75	61
roadNet-PA	6	610	4	+/-1740000
roadNet-CA	6	1741	5	+/-5400000
com-LiveJournal	360	69784	/	/

Table 2 – Résultats

5 Expériences sur des graphes issus de données réelles

5.1 Graphes

5.1.1 Social community

— ego-facebook : https://snap.stanford.edu/data/ego-Facebook.html
— LiveJournal : http://snap.stanford.edu/data/com-LiveJournal.html

5.1.2 Road network

Intersections and endpoints are represented by nodes, and the roads connecting these intersections or endpoints are represented by undirected edges.

— roadNet-PA : Pennsylvania.
— roadNet-CA : California.

5.2 Expériences

CCM : coefficient de connexion moyen

TMCD : Temps Moyen en milli-secondes de Calcul de la Dégénérescence

TMCC : Temps Moyen en milli-secondes de Calcul de la Colorabilité

5.3 Conclusions

Nous pouvons affirmer que, pour la majorité des graphes référencés sur SNAP, les algorithmes ont retourné le résulat demandé dans un temps raisonnable.

Le plus grand graphe que nous avons traité est "com-LiveJournal", un graphe de 4036538 sommets et 34681189 arêtes. Cependant, nous n'en avons pas fait la coloration, vu le temps que cela aurait pris.
En théorie, l'algorithme Wave Function Collapse est censé être extrêmement rapide, nous en concluant donc que notre implémentation, bien que donnant des résultats très précis, n'était pas la plus optimisée.

Le coefficient de connexion moyen est en moyenne dix fois plus élevé dans les graphes de communauté sociale que dans ceux des réseaux de routes. Ce rapport est confirmé dans nos résultats par ceux des dégénérescences d'un facteur allant de 100 à 300

6 Ressources

(cf. énoncé)

Wave Function Collapse Coloring : A New Heuristic for Fast Vertex Coloring :

https://www.researchgate.net/publication/354088959_Wave_Function_Collapse_Coloring_A_New_Heuristic_for_Fast_Vertex_Coloring

Théorème de Brook :

https://www.sciencedirect.com/science/article/pii/0095895675900891?via

Preuve du Lemme : https://math.stackexchange.com/questions/4430669/proof-suggestion-for-the-chromatic-number-of-a-k-degenerate-graph-is-inferior

Rédaction scientifique :

http://informatique.umons.ac.be/algo/redacSci.pdf.

Ressources bibliographiques :

https://www.bibtex.com/.

Classes de la bibliothèque Java algs4.jar, disponible à l'adresse suivante :

https://algs4.cs.princeton.edu/code/.

L'écriture du code a été accélérée à l'aide du plugin "Github Copilot" https://copilot.github.com/