



# S&R THEMIND

LACHAUD Samuel / PAZOLA Loïs – Info S6

# SOMMAIRE



## I) Introduction

- Qu'est-ce que GraphSharp
- Présentation de l'application



## II) Fonctionnement de l'application

- Affichage du graphe
- Calcul de dégénérescence
- Calcul du nombre chromatique



## III) Fonctionnalités supplémentaires

- Comparatif dégénérescence/nombre chromatique
- Génération d'un fichier PDF
- Algorithme de Matula & Beck

# I) Introduction

## Qu'est-ce que GraphSharp

Le projet S&R GraphSharp est un projet de graphe codé en C# portant sur le calcul de la dégénérescence de grands graphes réels. Ainsi à partir de deux algorithmes différents, nous allons pouvoir calculer ce nombre de dégénérescence et le comparé avec le nombre chromatique d'une multitude de graphes réels du projet KONNECT (<http://konect.cc/>). Nous proposerons également un export de la dégénérescence de chaque sommet des graphes en PDF sous formes de cercles imbriqués pour une visualisation rapide et démonstrative.

## Présentation de l'application

```
GraphSharp x + v
WELCOME TO OUR GRAPH PROJECT, CHOOSE AN OPTION IN THE LIST :
1. Display the graph
2. Calculate degeneration number
3. Calculate chromatic number
4. Generate a PDF with vertex degeneration
5. Compare degeneration and chromatic numbers for a lots of graphs
6. Calculate degeneration number with Matula Beck Algorithm
```

GraphSharp est une application console proposant par le biais du Windows Terminal, une interface basique demandant à l'utilisateur un choix parmi les différentes fonctionnalités.

Pour les actions autres que la comparaison des nombres chromatiques et de la dégénérescence, plusieurs graphes différents nous sont proposés avec des tailles différentes

```
GraphSharp x + v
CHOOSE A FILE :
1. Crime 12,4 Ko
2. ERoadNetworks 12,4 Ko
3. Exemple 86 o
4. Facebook 28,5 Ko
5. Flickr 100,6 Mo
6. GooglePlus 442,4 Ko
7. JazzMusicians 20,7 Ko
8. Physicians 8,6 Ko
9. PoliticalBooks 3 Ko
10. SisterCities 213,8 Ko
11. Twitter 377,4 Ko
12. Youtube 3,2 Mo
```

## II) Fonctionnement de l'application

### Affichage du graphe

Dans un premier temps, nous devons lire le fichier texte, ainsi, on détecte chaque ligne et on récupère les deux valeurs qui sont séparées par une tabulation (\t). Une fois récupéré, nous allons ranger ces valeurs de la manière suivante :

On crée un tableau composé de deux colonnes : la première étant le numéro du sommet et la deuxième une autre liste composée de tout les sommets qui ont une arrête en commun avec le sommet de la première colonne

```
static void ReadFile(string file)
{
    graph = new Dictionary<int, List<int>>();
    using (StreamReader streamReader = new StreamReader(@"Resources\" + file))
    {
        long fileLength = streamReader.BaseStream.Length;
        while (!streamReader.EndOfStream)
        {
            ShowProgression(streamReader.BaseStream.Position, fileLength);
            string readLine = streamReader.ReadLine();
            if (!readLine.StartsWith("%") && readLine != "") // read lines except lines who begin with "%"
            {
                string[] splitedLine = readLine.Split('\t'); // get the two ints of the line
                int curNode = Int32.Parse(splitedLine[0]);
                int nextNode = Int32.Parse(splitedLine[1]);
                Insert(curNode, nextNode);
                Insert(nextNode, curNode);
            }
        }
    }
}
```

Une fois le fichier chargé, l'affichage du graphe consiste juste à afficher le contenu du tableau nouvellement créé.

# Calcul de dégénérescence

Une fois que notre graphe est chargé dans notre tableau, nous pouvons calculer la dégénérescence de celui-ci.

```
static void VertexDegenerationFilling()
{
    vertexDegenerationTable = new Dictionary<int, int>();

    // copy graph
    Dictionary<int, List<int>> localGraph = graph.ToDictionary(entry => entry.Key, entry => new List<int>(entry.Value));
    int k = 0;
    int localGraphInitialSize = localGraph.Count;
    while (localGraph.Count != 0)
    {
        ShowProgression(localGraphInitialSize - localGraph.Count, localGraphInitialSize);
        // filling a list with all key we have to delete
        List<int> removeKeys = new List<int>();
        foreach (int key in localGraph.Keys) // for each keys
        {
            List<int> list = localGraph[key]; // get all nextNode of the key
            if (list.Count <= k)
            {
                removeKeys.Add(key);
            }
        }
        int removeKeysCount = removeKeys.Count;
        // Delete keys and delete values in nextnodes values
        foreach (int key in removeKeys) // for each keys
        {
            localGraph.Remove(key);
            foreach (int row in localGraph.Keys)
            {
                List<int> list = localGraph[row]; // get all nextNode of the key
                list.Remove(key);
            }
            vertexDegenerationTable.Add(key, k);
            ShowProgression(localGraphInitialSize - localGraph.Count, localGraphInitialSize);
        }
        if (removeKeysCount == 0)
        {
            k++;
        }
    }
    consoleWriter.Flush();
    Console.WriteLine("\nThe degeneration number is : " + k + "\n");
    DegenerationNumber = k;
}
```

Nous stockons le graphe précédemment chargé dans une copie afin de pouvoir localement supprimer les sommets déjà traités par la k-dégénérescence. Cela nous permet de ne pas altérer le graphe chargé et ainsi ne pas avoir à le recharger en mémoire durant une autre opération. En supprimant les nœuds au fur et à mesure, on obtient finalement la dégénérescence du graphe, le k le plus grand.

Sur l'exemple du cours, nous avons donc la dégénérescence  $k = 3$  qui est calculée :

```
GraphSharp x + v
-----
| Filling in the table from the file |
-----
[*****] 100%
-----
| Calculating degeneration number and filling the table |
-----
[*****] 100%
The degeneration number is : 3
```

## Calcul du nombre chromatique

Une fois que notre graphe est chargé dans notre tableau, nous pouvons calculer le nombre chromatique de celui-ci. Nous utilisons alors l'algorithme de DSATUR. Celui-ci est une transcription du programme vu en cours développé en C++, mais cette fois il est développé en C#.

Pour cela, nous devons changer notre structure comportant le graphe chargé, en effet pour le bon fonctionnement de l'algorithme DSATUR du cours, nous avons besoin d'une structure qui pour chaque sommet du graphe est associé un tableau où le numéro de ligne du tableau possède la valeur 1 si ce numéro de ligne (le numéro du sommet du graphe) est relié à celui-ci, et 0 si ce n'est pas le cas.

Pour le graphe de l'exemple nous obtenons donc une coloration en 3 couleurs. Pour ce qui est de la comparaison entre les différents graphes et le nombre de dégénérescence, nous en reparlerons dans la partie suivante.

```
GraphSharp x + v
-----
| Filling in the table from the file |
-----
[*****] 100%

-----
| Calculating chromatic number |
-----
4/4
[*****] 100%
DSAT Algorithm : Colouring in 3 colours.
```



# III) Fonctionnalités supplémentaires

Pour aller plus loin dans le projet, 3 améliorations ont été mises en place. Nous allons tout d'abord voir la comparaison entre la dégénérescence et le nombre chromatique sur un ensemble de graphe (**point 3.b du sujet**), puis la génération d'un fichier PDF (**point 3.c du sujet**) et enfin un meilleur algorithme pour le calcul de la dégénérescence qu'est l'algorithme de Matula & Beck (**point 3.a du sujet**).

## Dégénérescence/nombre chromatique

Après avoir calculé le nombre chromatique et la dégénérescence d'un graphe, nous pouvons donc maintenant la comparer sur l'ensemble des graphes présents dans le programme. En saisissant l'option 5 sur le programme, ces deux valeurs vont donc être calculées sur l'ensemble des graphes (sauf Flickr car beaucoup trop lourd). Un récapitulatif sera alors montré à l'utilisateur :

```
GraphSharp x + v
file : Crime
degenerationNumber = 3
chromaticNumber = 4

file : Exemple
degenerationNumber = 3
chromaticNumber = 3

file : Facebook
degenerationNumber = 3
chromaticNumber = 4

file : GooglePlus
degenerationNumber = 13
chromaticNumber = 8

file : JazzMusicians
degenerationNumber = 29
chromaticNumber = 30

file : Physicians
degenerationNumber = 9
chromaticNumber = 7

file : PoliticalBooks
degenerationNumber = 6
chromaticNumber = 6

file : SisterCities
degenerationNumber = 26
chromaticNumber = 27

file : Twitter
degenerationNumber = 10
chromaticNumber = 8

file : Youtube
degenerationNumber = 22
chromaticNumber = 10
```

A partir de ce rendu, nous pouvons donc déduire que l'affirmation « on peut colorier un graphe  $k$  dégénéré en au plus  $k + 1$  couleurs » est vrai dans notre cas d'observation. En effet la valeur **chromaticNumber** est toujours inférieure ou égale à la valeur **degenerationNumber** + 1.

On peut également supposer aux vues de ce rendu que l'écart entre la dégénérescence et le nombre chromatique est plutôt faible, mais que quelques exceptions sont tout de même présentes comme pour le fichier

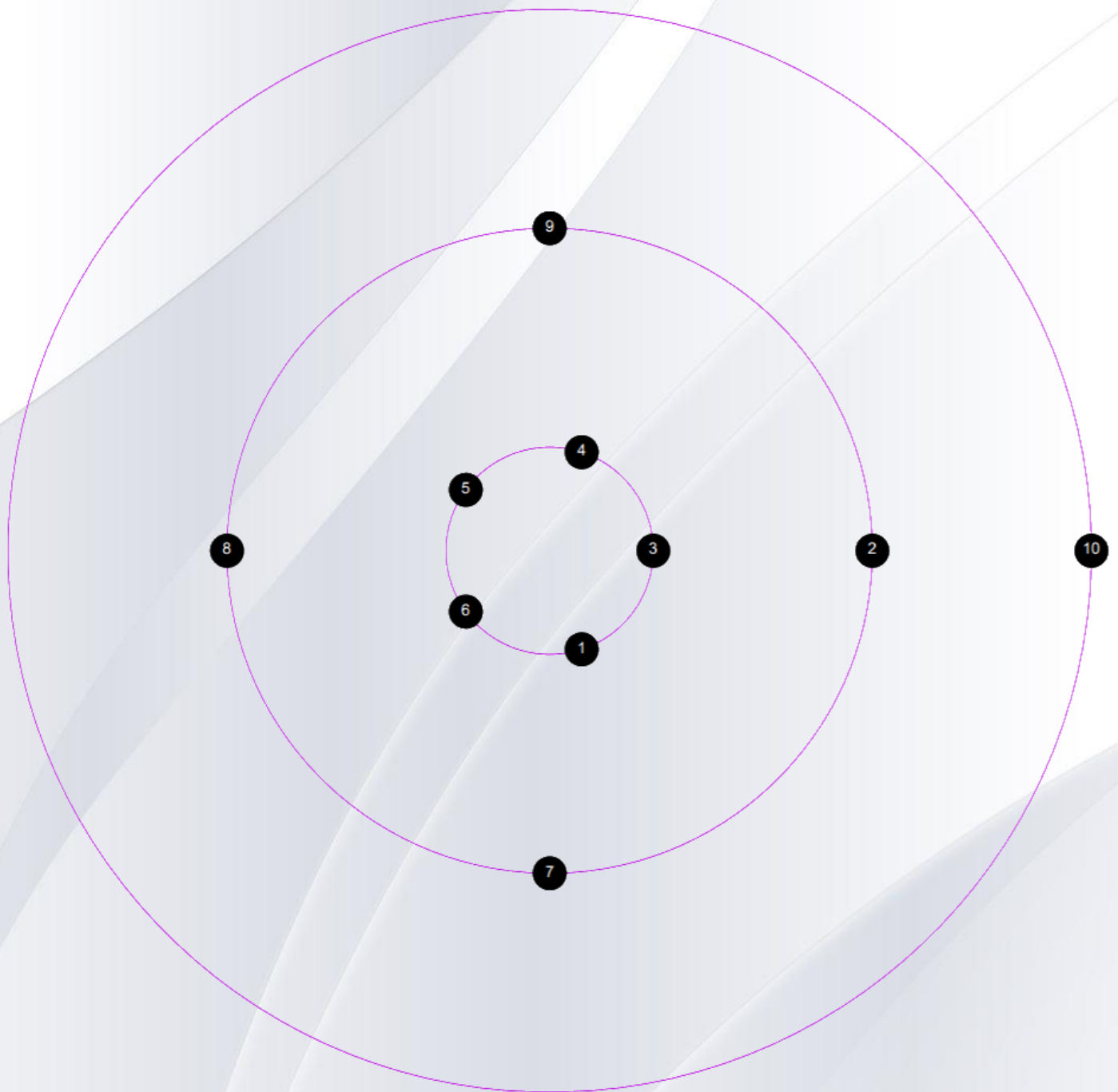
## Génération d'un fichier PDF

La dégénérescence de notre graphe est calculable, mais difficilement visualisable, en effet, on calcule la  $k$ -dégénérescence maximale, mais également la dégénérescence de chaque sommet du graphe. Nous pouvons donc visualiser ce graphe avec des cercles imbriqués, un cercle pour chaque  $k$ -dégénérescence. Le tout est mis dans un fichier PDF et chaque sommet est placé sur le cercle lui correspondant.

```
static void CreatePDF(string filename){
    string pdfFile = @"Resources\"+filename+".pdf";
    Document document = new Document(PaperSize.A0, 10, 10, 10, 10);
    FileStream fs = new FileStream(pdfFile, FileMode.Create, FileAccess.Write);
    PdfWriter writer = PdfWriter.GetInstance(document, fs);
    document.Open();
    // the pdf content
    PdfContentByte cb = writer.DirectContent;
    Console.WriteLine("-----");
    Console.WriteLine("| Create table k --> number of k-degeneration |");
    Console.WriteLine("-----");
    // create table k --> number of k-degeneration
    Dictionary<int, int> kNumbers = new Dictionary<int, int>();
    foreach (int key in vertexDegenerationTable.Keys.OrderBy(key => key)){
        if (!kNumbers.ContainsKey(vertexDegenerationTable[key])){
            kNumbers.Add(vertexDegenerationTable[key], 1);
        }
        else{
            kNumbers[vertexDegenerationTable[key]]++;
        }
    }
    Console.WriteLine("-----");
    Console.WriteLine("| PDF file creation |");
    Console.WriteLine("-----\n");
    int maxCircleSize = (int)(document.GetRight(-10) / 2) * 80 / 100;
    int minCircleSize = 150;
    foreach (int key in kNumbers.Keys.OrderBy(key => key)){ // circles for loop
        // circles creation
        int circleSize = maxCircleSize / DegenerationNumber * (DegenerationNumber - key) + minCircleSize;
        cb.SetColorStroke(new BaseColor(183, 3, 223));
        cb.Circle(document.GetRight(-10) / 2, document.GetTop(-10) / 2, circleSize);
        cb.Stroke();
        double angle = Math.PI * (360.0 / kNumbers[key]) / 180.0; // angle in rad
        int i = 0;
        foreach(int v in vertexDegenerationTable.Keys){
            if(vertexDegenerationTable[v] == key){
                // calculate the position of the little circles with initial position and angle
                double posX = (double)(document.GetRight(-10) / 2 + circleSize * Math.Cos(angle * i));
                double posY = (double)(document.GetTop(-10) / 2 + circleSize * Math.Sin(angle * i));
                double elemSize = 75 / DegenerationNumber;
                if(elemSize < 5){
                    elemSize = 5;
                }
                cb.Circle(posX, posY, elemSize);
                cb.SetColorFill(new BaseColor(0, 0, 0));
                cb.Fill();
                cb.SetColorStroke(new BaseColor(0, 0, 0));
                cb.Stroke();
                cb.BeginText();
                cb.SetColorFill(new BaseColor(255, 255, 255));
                cb.SetFontAndSize(BaseFont.CreateFont(BaseFont.HELVETICA, BaseFont.CP1252, BaseFont.NOT_EMBEDDED),(float)elemSize-2);
                cb.ShowTextAligned(Element.ALIGN_CENTER, v.ToString(), (float)posX, (float)(posY - elemSize / 5), 0);
                cb.EndText();
                i++;
            }
        }
    }
    document.Close();
    fs.Close();
    writer.Close();
}
```



Le programme ouvre donc le fichier PDF généré, voici celui correspondant à l'exemple. Ainsi le cercle extérieur représente les sommets de dégénérescence 1 puis à l'intérieur le 2 et enfin le 3 :



## Algorithme de Matula & Beck

La dernière amélioration de notre programme est l'ajout de l'algorithme de Matula & Beck. En effet il permet également de calculer la dégénérescence d'un graphe, mais de manière beaucoup plus rapide.


La mise en place de l'algorithme de Matula & Beck s'est fait à l'aide de la thèse suivante : [https://schulzchristian.github.io/thesis/thesis\\_huebner.pdf](https://schulzchristian.github.io/thesis/thesis_huebner.pdf).

```

static void VertexDegenerationFillingMatulaBeck(){
    //Initialize an output list L
    vertexDegenerationTableMatulaBeck = new Dictionary<int, List<int>>();
    //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
    Dictionary<int, int> d = graph.ToDictionary(entry => entry.Key, entry => entry.Value.Count);
    int BiggestI = d.Values.Max();
    int DInitialNumberOfElements = 0;
    //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
    List<int>[] D = new List<int>[BiggestI + 1];
    for (int j = 0; j <= BiggestI; j++){
        List<int> DiList = new List<int>();
        foreach (int key in d.Keys){
            if (d[key] == j){
                DiList.Add(key);
                DInitialNumberOfElements++;
            }
        }
        D[j] = DiList;
    }
    int k = 0; //Initialize k to 0
    int i = 0;
    bool DContainsValues = true;
    int DNumberOfRemovedElements = 0;
    //Repeat n times
    while (DContainsValues){
        DContainsValues = false;
        //Search for the lowest non-empty bucket
        if (D[i].Count == 0){
            for (int j = i; j < D.Length; j++){
                if (D[j].Count > 0){
                    i = j;
                    DContainsValues = true;
                    break;
                }
            }
        }
        else{
            DContainsValues = true;
        }
        k = Math.Max(k, i);
        if (DContainsValues){ //Remove the first vertex in the lowest non-empty bucket
            int v = D[i][0];
            D[i].Remove(v);
            DNumberOfRemovedElements++;
            if (vertexDegenerationTableMatulaBeck.ContainsKey(k)){
                vertexDegenerationTableMatulaBeck[k].Add(v);
            }
            else{
                vertexDegenerationTableMatulaBeck.Add(k, new List<int> { v });
            }
            d[v] = -1;
            foreach (int u in graph[v]){ //Update the neighbors of the removed vertex, after checking if they were not removed
                if (d[u] != -1){
                    if (i == d[u]){
                        i--;
                    }
                    D[d[u]].Remove(u);
                    d[u] -= 1;
                    D[d[u]].Add(u);
                }
            }
        }
    }
    consoleWriter.Flush();
    Console.WriteLine("\nThe degeneration number is : "+k+"\n");
}

```

Pour montrer cela, nous allons comparer la vitesse de calcul de la dégénérescence du graphe **Flickr**. Ce graphe est composé de plus de **8 500 000 arrêtes** et est stocké dans un fichier pesant près de **100Mo**. Le calcul est assez long et nous permettra de voir à quel point Matula & Beck est plus rapide :


GraphSharp

The degeneration number is : 187  
Execution Time: 11393796 ms

The degeneration number is : 187  
Execution Time: 269576 ms

<- Classic

<- Matula & Beck

11 393 796 / 269 576 = 42 x plus rapide